Université de Marne-La-Vallée

École Doctorale ICMS

ESIEE, Lab. A^{2}SI

THÈSE

pour obtenir le grade de

Docteur de l'Université de Marne-La-Vallée

Spécialité : Informatique

présentée et soutenue publiquement par

Linda KAOUANE

Formalisation et optimisation d'applications s'exécutant sur architecture reconfigurable

Directeur de thèse

Prof. Mohamed AKIL

Date de soutenance: le 17 Décembre 2004

Composition du Jury:

Président:	Jacques DESARMENIEN	Professeur, Univ. de Marne-la-Vallée, IGM
Rapporteur:	Jean Marc DELOSME	Professeur, Univ. d'Evry, LAMI
Rapporteur:	Habib MEHREZ	Professeur, Univ. de Paris6, LIP6-ASIM
Rapporteur:	Eduardo SANCHEZ	Professeur, EFPL-Lausane
Examinateur:	Mohamed AKIL	Professeur, ESIEE-Paris, Lab. A2SI
Examinateur:	Yves SOREL	Directeur de recherche, INRIA(Co-directeur de thèse)
Examinateur:	Thierry GRANDPIERRE	Enseignant-chercheur, ESIEE-Paris, Lab. A2SI

À mes chers parents

Remerciements

Je tiens tout d'abord à remercier Mohamed Akil, Professeur à l'ESIEE, et Yves Sorel, Directeur de recherche à l'INRIA-Rocquencourt, d'avoir partagé la tâche d'encadrer ce travail.

Je tiens à remercier monsieur Jacques Desarmenien, professeur et directeur de l'institut Gaspard Monge de l'Université de Marne-La-Vallée de m'avoir fait l'honneur de présider ce jury.

Je voudrais aussi exprimer ma gratitude à messieurs Jean Marc Delosme, professeur au laboratoire de Méthodes Informatiques (LAMI) de l'université d'Evry, Habib Mehrez professeur au laboratoire d'Informatique de l'université de Paris 6 (LIP6) et Eduardo Sanchez professeur au département Informatique, laboratoire LSL de l'Ecole Polytehnique de Lausanne d'avoir accepté la lourde tâche de rapporter sur cette thèse.

Enfin, pour son aide et sa participation au jury je remercie monsieur Thierry Grandpierre, enseignant chercheur à l'ESIEE.

Puisque l'occasion m'en est donnée, je veux aussi exprimer toute ma reconnaissance envers mes plus proches collègues Gamal Attiya, Pierre Niang, Cederic Sibade, Benoît Kaufmann, Fabiane Delazuna et Eric Lorens, Laurant Perroton et envers tous les autres membres du laboratoire A^{2}SI pour l'aide qu'ils m'ont apporté au cours de ces années.

Ces remerciements ne seraient pas complets sans mes pensées pour les êtres qui me sont les plus chers : mes parents, qui m'ont beaucoup appris, et qui, surtout, ont cru en moi qu'ils trouvent ici l'expression de ma reconnaissance.

Linda KAOUANE
Décembre 2004,
Paris, France.

Résumé

AAA (Algorithme Adéquation Architecture) est une méthodologie développée pour le prototypage rapide d'applications dédiées et temps réel. Basée sur un formalisme de graphes et un flot de conception unifié sans rupture, cette méthodologie aide le développeur à implanter et optimiser l'implantation d'algorithmes de traitements de signal et d'image sur des architectures multicomposants en supportant toutes les étapes du développement depuis la spécification algorithmique jusqu'au choix de l'implantation qui respecte les contraintes temps réel et l'exécution du code correspondant dans les composants.

Toutefois le flot de l'implantation actuellement supporté par cette méthodologie est limité aux composants cibles de types processeurs. Et comme l'exécution temps réel de nos applications cibles exige souvent une utilisation conjointe de composants processeurs et de composants circuits spécialisés, la contribution de ce travail consiste à étendre la méthodologie AAA vers des architectures circuits. Étape jugée indispensable pour utiliser pleinement la méthodologie AAA pour faire la conception conjointe où la cible multiprocesseurs coopère avec une cible câblée.

Dans cette thèse, nous nous sommes donc intéressés à l'extension de la méthodologie AAA pour le support des implantations sur de composants circuits reconfigurables. Nous avons défini et formalisé une méthodologie et un flot de prototypage rapide supportant cette extension depuis la spécification algorithmique sous forme de graphe factorisé et conditionné de dépendances de données (GFCDD) jusqu'à la génération du code RTL décrivant l'architecture circuit de l'implantation matérielle.

Ce flot a été implanté dans SynDEx-IC, logiciel d'aide au prototypage rapide spécialement développé à partir du noyau de l'environnement SynDEx supportant l'AAA-multiprocesseurs pour supporter cette extension d'AAA.

Mots-clés : *Adéquation Algorithme-Architecture, Systèmes temps réel, Prototypage rapide, Synthèse de circuits, Optimisation, Transformations de graphes, Génération automatique de code.*

Abstract

The AAA (Algorithm-Architecture Adequation) is a methodology developed for rapid prototyping of dedicated real-time applications. Based upon formal graph model and unified seamless flow, this methodology helps the designer to implement and optimize signal and images processing algorithms onto multicomponent architectures. It supports all the design steps from algorithm specification until the choice of the optimized implementation that satisfy the real-time constraints and code execution on the components.

However the flow supported by this methodology is currently limited to multiprocessor componenents. And Since the real-time execution of our applications often demands a joint use of processors components and specialized circuits components, the contribution of this work is to extend the AAA methodology for the use of circuits architectures. Essential step to fully use the AAA methodology to do codesign where the multiprocessors components cooperates with the dedicated ones.

In this thesis, we are interested in the extension of the AAA methodology for the support of the implementation onto reconfigurables circuits components. We so defined and formalized a methodology and a design flow of rapid prototyping supporting this extension since the algorithmic specification in term of a factorized and conditioned data dependency graph (GFCDD) to the generation of the RTL code describing the circuit architecture.

This flow has been implemented in SynDEx-IC, a CAD software tool, that has been developed from the kernel of the SynDEx environment which support the AAA-multiprocessors.

Keywords : *Algorithm-Architecture Adequacy, Real-time systems, Rapid prototyping, Circuits synthesis, Optimizations, Graph transformations, Automatique code generation.*

Table des Matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Ce premier chapitre d'introduction présente le contexte général de cette thèse, la problématique traitée et donne un aperçu global de la démarche suivie et du travail effectué. Pour conclure ce chapitre, nous donnons l'apport de cette thèse et le plan de ce manuscrit.

1.1 Contexte : Systèmes réactifs temps réel embarqués

Nos recherches concernent le développement d'outils d'aide à l'implantation efficace des applications de contrôle/commande et de traitement du signal et des images, soumises à des contraintes temps réel et d'embarquabilité. Ces applications qualifiées d'applications réactives-temps réel embarquées désignent habituellement des systèmes composés d'un couple de deux sous-systèmes en constante interaction. Le premier sous-système correspond à l'environnement physique de l'application, dont les changements d'état, doivent être contrôlés en permanence. Le second correspond à un système réactif qui doit réagir aux variations d'états du premier, lui même composé d'un ensemble d'algorithmes codant l'application et d'une architecture matérielle supportant l'exécution de ces algorithmes issus principalement du domaine du contrôle-commande et du traitement du signal.

Le terme réactif qualifie ainsi le comportement de ces systèmes alors que le terme temps réel est relatif aux contraintes qui leur sont imposées. En effet, ces systèmes sont qualifiés de réactifs puisqu'ils doivent réagir immédiatement aux événements d'entrée provenant de l'environnement (stimulus) en produisant des événements en sorties (réactions) utilisables par cet environnement. De plus, ils sont qualifiés de temps réel puisqu'ils sont soumis à un ensemble de contraintes temps réel afin d'assurer que l'environnement est bien contrôlé. Il s'agit principalement dans notre cadre d'étude, des contraintes de latence (temps maximum de réaction entre la variation des entrées et la réaction correspondante) ou de cadence (durée entre deux acquisitions d'une même entrée). Enfin, la plupart de ces systèmes sont également soumis à des contraintes technologiques d'embarquabilité et de coût, c'est à dire que les ressources matérielles mises en jeu doivent être minimisées en terme de consommation, de coût, de poids, de prix, ...

De nos jours, cette classe d'applications réactives-temps réel embarquées, principalement ciblées par nos activités de recherche, occupe une place de plus en plus importante dans le monde qui nous entoure. On les retrouve maintenant aussi bien dans les produits grand public (automobile, téléphone, équipements hifi et audio) que dans les équipements industriels lourds (centrale nucléaire, chaîne de fabrication, avionique, systèmes d'armes, aéronautique). L'importance prise par ce type de systèmes a amplifié considérablement le besoin de développement de nouvelles méthodes et d'outils logiciels d'aide à l'implantation intégrant l'ensemble des étapes depuis la spécification haut niveau jusqu'à la mise en oeuvre de l'implantation qui respecte les contraintes temps réel et minimise les ressources matérielles utilisées tout en réduisant les temps de développement.

Ce mouvement a été motivé par la complexité toujours croissante de ces applications et le coût élevé de leurs implantations, notamment dans un contexte temps réel. Cette complexité résulte, entre autres, à la fois de l'augmentation du nombre de fonctionnalités mises en oeuvre mais aussi de l'éventail des possibilités d'architectures fournies par la technologie. En effet, avec la diversité des solutions logicielles et/ou matérielles potentielles fournies par les progrès de la technologie, les concepteurs sont de plus en plus confrontés au difficile problème de déterminer les meilleures architectures de systèmes qui implémentent les fonctionnalités requises de telle sorte que les contraintes de temps et de surface propres à l'application soient respectées. D'autre part, avec les progrès réalisés dans le domaine du traitement du signal et des images, les applications utilisent des algorithmes de plus en plus sophistiqués dont la complexité ne cesse de croître.

Bien évidemment, cette augmentation de complexité, tant au niveau algorithmique que matériel, entraîne une probabilité accrue d'erreurs de conception et rend le processus d'implantation un exercice

délicat à maîtriser, ce qui ne fait que croître les coûts de conception et de mise sur le marché de ces systèmes.

Il devient donc nécessaire de définir de nouvelles méthodes permettant de concevoir de manière sûre ces systèmes, et d'obtenir une implantation efficace à partir de spécifications tout en minimisant la durée du cycle conception-implantation. De telles méthodes permettent donc aux concepteurs de se concentrer sur les aspects temporels qui sont cruciaux dans le domaine du temps réel (réactivité du programme et temps de réponse contraint), d'étudier les relations entre le parallélisme potentiel au niveau de l'algorithme de l'application et celui disponible au niveau de l'architecture matérielle distribuée et par conséquent de consacrer plus leurs temps à la spécification des applications et non plus à la phase d'implantation souvent fastidieuse.

Pour répondre à ces besoins, plusieurs approches de conception ont émergé ces dernières années. La conception conjointe du logiciel et du matériel (ou Codesign) était l'une des plus explorées par les groupes de recherche du fait que l'exécution temps réel des ces applications exige souvent une utilisation conjointe de processeurs, et de circuits spécialisés. Il s'agit de méthodologies et d'outils de CAO qui couvrent les étapes de spécification, de validation et d'exploration des différentes alternatives d'implantation (matériel/logiciel) de ces applications, dans le but d'optimiser les critères de coût et/ou de performances qui leurs sont imposés.

Il s'agit habituellement d'approches visant à gérer au mieux l'hétérogénéité de l'architecture cible dans le cadre d'un environnement de conception intégré qui permet la synthèse automatique des implantations matérielles, des implantations logicielles, et de la communication en partant d'une spécification de haut niveau du système complet et en intégrant les techniques de conception du matériel et du logiciel dans un processus unique, structuré et automatique.

Généralement, dans de telles approches les systèmes sont d'abord spécifiés, puis vérifiés et enfin optimisés. La spécification d'un système consiste souvent à décrire l'algorithme de l'application, l'architecture, et l'implantation de l'algorithme sur l'architecture. Du fait de la diversité des applications cibles et par souci de portabilité et de réutilisabilité des algorithmes et architectures, ces approches de conception conjointe s'orientent, de nos jours, de plus en plus vers des approches globales basées sur une séparation nette entre les deux vues : fonctionnelle (algorithmes décrivant le comportement de l'application), architecturale (architecture matérielle supportant leur implantation) dès les premières étapes du cycle conception-implantation. Ce modèle de construction particulier (appelé spécification en "Y" "*Y-chart model*") repose sur une spécification séparée des algorithmes de l'application, de la spécification de l'architecture supportant leur implantation, de l'étape d'implantation proprement dite de ces algorithmes sur une architecture particulière. Cette séparation des modèles autorise par construction aussi bien le portage des algorithmes et architectures que leur réutilisation : une même application peut être réutilisée sur une nouvelle architecture, ou être transformée et ré-implantée sur une architecture existante. Ce qui favorise une exploration efficace de l'espace de conception en vue de la recherche de la meilleure adéquation entre algorithmes et architectures. Cette exploration efficace réduit considérablement le délai de la recherche de la meilleure implantation (adéquation algorithme-architecture) et, par là même, une arrivée plus rapide du produit sur le marché. Par adéquation, nous entendons ici la mise en correspondance de manière efficace de l'algorithme et de l'architecture pour réaliser une implantation optimisée.

C'est pourquoi, depuis quelques années, la tendance du monde industriel et de la recherche est

de proposer de nouvelles méthodologies de conception conjointe basées sur une telle construction. En fait, les méthodologies basées sur de telles approches permettent d'aborder de manière plus claire le problème de la conception conjointe logiciel-matériel en améliorant leur capacité à faire d'une part le prototypage rapide et d'autre part à obtenir les implantations respectant les contraintes temps réel tout en minimisant l'architecture matérielle.

1.2 Cadre de l'étude : La méthodologie AAA d'adéquation algorithme-architecture

Comme nous venons de le mentionner ci-dessus, il devient de plus en plus indispensable dans un processus de codesign de pouvoir aborder la conception sous l'angle de l'exploration, l'estimation et le prototypage, en développant des méthodologies de conception qui privilégient l'adéquation entre les algorithmes des applications visées et les architectures matérielles supportant leur implantation. Avant tout, nous allons préciser d'abord cette notion d'adéquation algorithme-architecture.

En fait, la problématique de l'adéquation algorithme-architecture est en général un problème d'optimisation complexe visant à mettre en correspondance de manière efficace l'algorithme et l'architecture pour réaliser une implantation optimisée. Ceci conduit globalement à étudier simultanément les aspects algorithmiques et architecturaux afin de déterminer quelle est l'architecture d'un système complet qui est la mieux adaptée à un traitement et, à l'inverse, comment reformuler l'algorithme de l'application, voire le transformer, pour faciliter son implantation optimisée.

Plus précisément, le problème se pose d'abord de trouver l'architecture la mieux adaptée à un algorithme donné satisfaisant à certaines contraintes. On choisit ainsi une solution logicielle programmée en utilisant un ou plusieurs processeurs ou/et une solution matérielle câblée sous la forme d'un ASIC ou/et de un ou plusieurs FPGA. Cependant, lorsque l'on ne peut pas de cette manière satisfaire les contraintes, on est parfois conduit à modifier l'algorithme lui-même. On doit alors soit modifier sa granularité, ce qui amène simplement à un redécoupage de l'algorithme, soit modifier plus radicalement sa structure (reformulation complète). Ce processus itératif dans lequel l'algorithme influence l'architecture et réciproquement est l'essence même de l'adéquation algorithme architecture.

En se basant, sur ce concept d'Adéquation Algorithme Architecture (démarche), l'équipe OSTRE de l'INRIA-Rocquencourt (Institut National de Recherche en Informatique et en Automatique) a développé une méthodologie formelle d'adéquation algorithme-architecture, nommée AAA, particulièrement adaptée pour les systèmes temps réel embarqués. Cette méthodologie de prototypage rapide et d'implantation optimisée, est le cadre principal où se développe notre travail, c'est pourquoi nous rappelons brièvement par la suite ses principes.

Les principes de cette méthodologie, conçue sur un modèle de construction en "Y", sont la spécification de l'algorithme à l'aide d'un graphe de dépendances de données, mettant en évidence le parallélisme potentiel de l'algorithme, et la spécification de l'architecture à l'aide d'un graphe de ressources de calcul, de ressources mémoire et de ressources de communication, mettant en évidence le parallélisme disponible ; puis l'implantation exprimée en terme de transformations de graphes, c'est-à-dire distribution et ordonnancement de l'algorithme sur l'architecture multicomposants et, enfin, génération automatique de l'exécutif temps réel distribué correspondant à l'implantation optimisée de l'algorithme spécifié sur l'architecture donnée.

Étant basée sur une modélisation unifiée (à base de graphes) aussi bien de l'algorithme, et de l'architecture que de l'implantation, cette méthodologie offre un cadre formel et un flot de conception unifié sans rupture permettant d'une part de faire des vérifications temporelles en termes d'ordre sur les événements qui entrent et sortent de l'algorithme, éliminant ainsi un grand nombre d'erreurs concernant la logique de l'algorithme. D'autre part d'aider à la recherche, parmi toutes les implantations possibles que l'on peut faire d'un algorithme, de l'implantation optimisée qui respecte les contraintes temps réel et minimise les ressources matérielles. Et enfin de générer automatiquement des exécutifs taillés sur mesure pour l'application, sans interblocage et dont le surcoût est très faible, en assurant que les vérifications faites précédemment restent valides. Puisque le code embarqué est correct par construction, la phase du test de code peut être éliminée et l'application prototype obtenue ainsi peut être directement utilisable comme produit de série. Tout ceci place AAA ainsi que le logiciel SynDEx (acronyme de *Synchronized Distributed Executif* – exécutif distribué synchronisé) qui la supporte parmi les méthodes et outils logiciels d'aide à l'implantation les plus avancés dans le domaine de développement des applications temps réel embarquées mettant oeuvre des algorithmes de contrôle-commande comprenant du traitement du signal et des images.

Toutefois, le flot d'implantation supporté par cette méthodologie est limitée aux architectures cibles de types multiprocesseurs, la contribution de ce travail consiste à étendre son utilisation vers des architectures mixtes câblées et programmées.

1.3 Motivations et objectifs : Extension d'AAA vers le Co-design

Face aux exigences de plus en plus fortes des contraintes temps réel et d'embarquabilité, les systèmes temps-réel embarqués sont contraints à utiliser en complément des processeurs, des circuits intégrés spécialisés (ASIC figés ou FPGA reconfigurables).

Comme AAA cible des systèmes nécessitant souvent une utilisation conjointe de processeurs et de circuits intégrés spécialisés afin de satisfaire les contraintes imposées, il s'est avéré naturel d'étudier son utilisation pour la conception conjointe où la cible multiprocesseur coopère avec une cible matérielle câblée.

Ce travail de thèse, en continuité avec les travaux précédents menés par l'équipe A2SI de l'ESIEE en collaboration avec le projet OSTRE visant cet objectif, se propose d'étendre l'utilisation de AAA pour des architectures circuits. Cette étape est jugée indispensable pour utiliser pleinement la méthodologie AAA pour faire de la conception conjointe matériel/logiciel.

L'objectif général encadrant cette thèse consiste donc à proposer un flot de conception supportant l'extension de la méthodologie aux implantations circuits. Ce flot se propose :

- de décrire la spécification de l'algorithme de l'application en prenant en compte aussi bien les aspects flot de données que ceux concernant le flot de contrôle,
- de définir et d'automatiser l'ensemble des transformations nécessaires pour aboutir à l'implantation finale,
- de formaliser ces transformations à l'aide d'un formalisme commun issu de la théorie des ensembles et des graphes en vue de préserver la cohérence des différentes parties du système, à travers toutes les étapes du processus d'implantation, et de garantir ainsi une implantation valide par

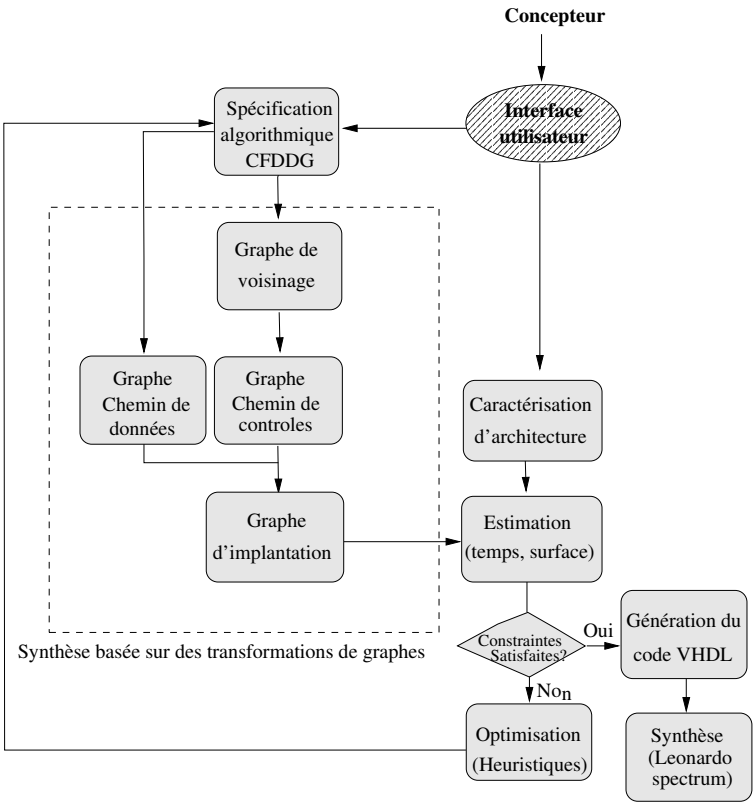


FIG. 1.1 – Flot d'implantation sur circuits d'AAA

construction,

- de permettre l'exploration à l'aide d'heuristiques de l'espace des implantations possibles en se basant sur une estimation de haut niveau des caractéristiques de l'implantation,
- de générer automatiquement le code VHDL structurel et synthétisable correspondant à l'implantation matérielle optimisée retenue par l'heuristique.

Ce flot continu de conception, illustré figure 1.1, consiste en une séquence de tâches qui transforme un modèle de spécification en une implantation. Le graphe modélisant l'algorithme est transformé progressivement jusqu'au graphe d'implantation matérielle modélisant l'architecture au niveau RTL du circuit visé. Nos travaux s'articuleront principalement autour de l'expression du contrôle dans le modèle flot de données utilisé pour la spécification algorithmique, de la formalisation et l'automatisation de ces différentes transformations, ainsi que de l'optimisation de l'implantation finale et la mise en oeuvre d'une génération automatique du code VHDL correspondant.

Comme le point de départ du processus d'implantation matérielle d'une application sur une architecture est la spécification algorithmique, nous nous sommes intéressés dans un premier temps à l'amélioration du modèle de spécification utilisé pour prendre en compte l'ensemble du contrôle, la partie itérative ayant été traitée dans la thèse de Ailton Dias [27]. En effet, si le modèle orienté flot de données est parfaitement adapté pour spécifier le parallélisme potentiel inhérent à l'algorithme de l'application spécifiée, il ne permet pas de prendre en compte les structures de sélection et de répétition

liées au flot de contrôle et qui sont nécessaires à la spécification de l'algorithme d'application. Les applications complexes imposant un mélange de traitement de données et de contrôle il est donc nécessaire de pouvoir faire cohabiter flot de contrôle et flot de données. Nous utilisons donc un modèle de spécification, appelé graphe factorisé et conditionné de dépendance de données (GFCDD), qui permet d'introduire du flot de contrôle à l'intérieur du flot de données dans un même modèle de spécification. L'objectif de cette combinaison est donc de garder globalement un modèle flot de données avec ses avantages (en particulier le parallélisme), et en même temps d'offrir des structures de contrôle (sélection : "if...then..else", répétition "For i=1 to N Do..."). Ce qui donne alors une souplesse d'utilisation puisque le concepteur peut spécifier en termes de flot de données et/ou en termes de flot de contrôle sans avoir besoin de changer de modèle.

À partir de cette spécification en termes de GFCDD, la technique d'implantation que nous préconisons dans cette extension d'AAA consiste en une suite de transformations progressives de ce graphe d'algorithme ($G_{al} = (O, D)$) afin d'obtenir un graphe d'implantation ($G_{im} = (O", D")$) comprenant aussi bien le chemin de données ($G_{cd} = (O"_1, D"_1)$) que de contrôle ($G_{cc} = (O"_2, D"_2)$) du circuit correspondant ($G_{im} = G_{cd} \cup G_{cc}$). Alors que les transformations nécessaires à la génération du chemin de données G_{cd} sont simples et systématiques, les transformations nécessaires à la génération du chemin de contrôle G_{cc} ne sont pas aussi simples. Ceci nous a amené à produire un graphe intermédiaire résultat d'une première transformation que nous appelons graphe de voisinage $G_v = (O', D')$. Ce graphe obtenu automatiquement à partir du graphe algorithmique, modélise à un niveau d'abstraction très élevé aussi bien les structures de sélection que de répétition présentes dans le graphe algorithmique ce qui nous permettra d'établir les relations de contrôle de manière simple et systématique en suivant des règles simples de synthèse basées sur le modèle RTL et des mécanismes de transferts de données synchronisés. Par souci de traçabilité et de validité de l'implantation finale, on s'est attaché tout au long de ce processus d'implantation à modéliser ces transformations à l'aide d'un formalisme commun issu de la théorie des ensembles et des graphes [124]. L'ensemble de ces transformations garantit donc par construction une implantation valide du graphe d'algorithme sur une architecture circuit.

Comme l'objectif recherché est de choisir, parmi l'ensemble des implantations valides, l'implantation optimisée qui respecte les contraintes temporelles et minimise la consommation de ressources, une comparaison automatique des performances (latence, cadence, quantités de ressources requises) des différentes implantations possibles est donc indispensable et surtout elle doit se faire sans que l'utilisateur n'ait à les réaliser. En effet, réaliser chaque implantation possible pour en mesurer les performances est inabordable : le nombre de solutions est très important, la durée de la réalisation d'une implantation peut être longue, et il se peut que l'architecture ne soit pas encore disponible au moment où l'on veut faire la comparaison. La comparaison systématique par la mesure étant inabordable, il est donc nécessaire de construire un modèle prédictif de performances (temps, surface) des implantations très précis et fiable afin de guider de manière efficace l'exploration de l'espace de solutions par l'algorithme d'optimisation d'implantation. Pour se faire, nous avons défini un modèle prédictif d'estimation de performances globales, appelé modèle de caractérisation, indépendant du type de circuit, basé sur une composition des caractéristiques des opérateurs (surface, temps de réponse), obtenue par les outils de simulation appliqués à chaque opérateur isolé.

Suite à cette estimation de performances, si l'implantation courante ne respecte pas les contraintes temps réel, nous appliquons un processus de défactorisation (transformation spatiale/temporelle

équivalente à un déroulage de boucles) de la spécification afin de tirer profit du parallélisme des calculs et d'obtenir une implantation plus parallèle, ce qui améliore les performances temporelles au prix de ressources matérielles supplémentaires. Parmi toutes les implantations défactorisées possibles, on éliminera celles qui ne respectent pas les contraintes temps réel, et on choisira celle qui minimise les ressources nécessaires à l'implantation tout en respectant les contraintes temporelles. On est face ainsi à un problème d'optimisation (recherche d'un minimum sous contraintes) connu pour être NP-difficile, dont la taille est généralement grande pour des applications réelles à cause de la complexité combinatoire exponentielle de la recherche de la solution exacte (optimale). C'est pourquoi on se contente seulement d'une solution approchée obtenue à l'aide d'heuristiques. Guidées par une fonction de coût permettant la comparaison des performances des différentes implantations, ces heuristiques permettent d'explorer seulement un sous-ensemble de toutes les implantations défactorisées possibles. Étant donné que nous nous intéressons au prototypage rapide nous avons privilégié dans un premier temps les "heuristiques gloutonnes" qui sont très rapides et donnent des résultats de bonne qualité. L'heuristique gloutonne proposée consiste à appliquer des transformations de défactorisations en évaluant, à chaque transformation à l'aide de la fonction de coût, si les contraintes sont respectées. Plus concrètement la fonction de coût calcule la latence du circuit et sa surface relativement à une caractérisation basée sur notre modèle d'estimation de performances et exprimée en termes de durée et de surface des éléments de bibliothèque VHDL synthétisables selon la technologie du circuit intégré visé. Il est tout de même intéressant de noter que nous avons développé aussi des heuristiques de voisinages de type "recuit simulé" fondées sur le même type de fonction de coût, beaucoup plus lentes, mais plus précises [125][127].

Le but recherché étant de générer automatiquement l'architecture circuit supportant l'implantation matérielle optimisée de l'algorithme spécifié, nous avons défini un ensemble de règles permettant de générer automatiquement et de manière systématique le code VHDL synthétisable correspondant à l'implantation optimisée, une fois que celle-ci a été déterminée par l'heuristique d'optimisation. Cette génération consiste en un processus à deux étapes, dans lequel nous procédons à la génération d'un macro code m4 avant de transformer ce dernier en code VHDL à partir d'une librairie VHDL que nous avons définie (VHDLlib.m4). Ce choix de génération de code intermédiaire en vue de la génération du code VHDL final a été motivé par souci d'indépendance du générateur vis à vis du langage cible qui peut être aussi Verilog ou SystemC par exemple. Cette extension du flot d'implantation dans la méthodologie AAA a été validé à travers un certains nombres d'exemples de traitement de signal et d'images illustrant parfaitement les problèmes liés aux implantations temps réel tels que : filtre moyenneur, le filtre de détection de contour de Dériche [126][128].

Enfin, ce flot de génération automatique d'implantation matérielle optimisée a été intégré dans un outil logiciel comportant une interface graphique permettant de spécifier des algorithmes selon le même modèle que dans le logiciel SynDEx et une heuristique d'optimisation et un générateur automatique de code VHDL structurel synthétisable. Cet outil nommé *SynDEx-IC* conçu spécialement pour supporter notre extension d'AAA est disponible sur le web à l'url : http ://www.esiee.fr/a2si/syndex-ic.

1.4 Structure de la thèse

Ce mémoire de thèse débute par deux chapitres d'état de l'art consacrés l'un au domaine de la conception conjointe en général, et l'autre à la méthodologie AAA d'adéquation algorithme-architecture. Les

chapitres suivants seront consacrées au développement de notre flot d'extension d'AAA. Chaque chapitre sera ainsi consacré à une étape particulière du flot.

Le chapitre 4 concerne le point d'entrée de notre flot spécification-implantation qui est la spécification algorithmique. Après avoir réalisé un état de l'art des différents modèles de spécification, nous présentons le modèle de spécification algorithmique et nous faisons le point sur notre enrichissement du modèle par l'introduction et l'intégration du contrôle de conditionnement dans le modèle de base. Enfin, ce modèle cohabitant flot de contrôle et flot de données sera formalisé en termes de théorie de graphes permettant ainsi de mener les preuves de la préservation de l'ordre partiel initial suite aux différentes transformations lors de l'implantation finale.

Le chapitre 5 traite la deuxième étape du flot qui est l'implantation matérielle. Après un état de l'art sur les stratégies et outils de synthèse comportementale, nous nous plaçons dans le contexte et définissons notre modèle d'implantation. Plus précisément, on construit en intention, l'ensemble des implantations valides d'un algorithme en appliquant un ensemble de transformations de graphes qui permettent de synthétiser le chemin de données et de contrôle d'une spécification flot de données. L'approche préconisée est capable de synthétiser toute la partie de contrôle des parties répétées du graphe (boucles) et des parties conditionnées. Le contrôle généré est un contrôle réparti plutôt qu'un contrôle centralisé ce qui permet de rapprocher les unités dépendantes et ainsi minimiser les surcoûts dûs au routage.

Le chapitre 6 décrit comment, parmi l'ensemble des implantations valides construites selon le modèle d'implantation présenté dans le précédent chapitre, on sélectionne celle qui respecte les contraintes imposées. Pour cela nous avons développé des heuristiques d'optimisation basées sur la défactorisation, c'est à dire dérouler plus ou moins les boucles de façon à trouver pour chaque boucle le taux de défactorisation (taux de déroulage) qui permet de respecter les contraintes d'exécution données.

Le chapitre 7 explicite la démarche suivie pour la génération du code RTL décrivant l'architecture du circuit d'implantation, une fois que l'implantation optimisée a été déterminée par l'heuristique d'optimisation. L'approche proposée est basée sur une génération d'un macro code m4 intermédiaire permettant d'assurer une indépendance du processus de génération vis à vis du langage de description cible qui peut être aussi n'importe quel langage de description en matériel : VHDL, Verilog ou HardwareC par exemple.

Le chapitre 8 présente l'application de notre flot à la conception des circuits dédiés reconfigurables de type FPGAs. Enfin, le chapitre 9 présente le logiciel d'aide au prototypage rapide d'applications temps réel SynDEx-IC développé spécialement pour supporter le flot d'implantation proposé. C'est un logiciel libre qui permet la synthèse automatique de circuits ASIC et de circuits reconfigurables de type FPGA.

Chapitre 2

La conception conjointe de systèmes mixtes matériel/logiciel

Le but de ce chapitre est de présenter l'état de l'art des systèmes de conception matériel/logiciel. Nous décrivons tout d'abord l'activité de conception conjointe matériel/ logiciel appelée usuellement co-design, son domaine d'application et les raisons de l'émergence de ce cycle de développement, ses principaux concepts et étapes. Après nous passons en revue quelques systèmes représentatifs de la conception conjointe. Les caractéristiques de chaque système seront détaillées afin de montrer les domaines d'application respectifs. Cette introduction permettra au lecteur d'appréhender le contexte actuel de la conception conjointe afin de mieux pouvoir situer la méthodologie AAA présentée dans le chapitre suivant et le travail de son extension vers le co-design.

2.1 Introduction

Pour faire face à la complexité croissante des applications temps réel et pour répondre aux critères de performances et de coûts attendus, il est souvent indispensable de faire recours lors de l'implantation à des architectures qui ne contiennent pas seulement des composants programmables (processeurs d'usage général, micro-contrôleurs ou DSP – *Digital Signal Processor*) mais aussi de composants non programmables (circuits spécialisés : ASIC – *Application Specific Integrated Circuit*, FPGA – *Field Programmable Gate Array*, IC – *Integrated Circuit* ou PAL –*Programmed Array Logic*). Seule une architecture mixte multicomposants est en effet susceptible de satisfaire les contraintes temps-réel imposées tout en répondant le mieux aux critères de coûts attendus en permettant la considération de plusieurs possibilités d'implantation : une fonctionnalité donnée peut ainsi être implantée de diverses manières, chacune avec ses propres caractéristiques de coût et performance.

Naturellement, l'implantation sur une cible logicielle d'un sous-ensemble de l'application permet d'obtenir rapidement, donc à moindre coût, l'implantation finale et offre plus de flexibilité (le même processeur peut implanter différentes applications en exécutant des codes différents), mais les performances peuvent être limitées pour certaines applications. Par contre l'implantation sur une cible matérielle permet de concevoir une architecture spécialement dédiée à l'implantation de la fonctionnalité désirée et donc d'obtenir de très bonnes performances, mais nécessite en contre partie un temps et un coût de conception supérieure à une implantation logicielle.

Dès lors, pour concevoir un système temps réel d'un coût raisonnable tout en respectant les contraintes temps réel imposées, on s'oriente de plus en plus vers des systèmes mixtes matériel/logiciel. L'implantation de ces applications temps-réel doit donc réaliser un compromis entre les avantages et les inconvénients des implantations matérielles et logicielles afin de bénéficier à la fois de la performance (rapidité d'exécution) de l'implantation en matériel, mais aussi du faible coût et de la flexibilité de l'implantation en logiciel.

Cette utilisation conjointe de ressources (composants) logicielles et matérielles nécessite le développement de nouvelles méthodologies de conception afin de résoudre les problèmes de conception simultanée matériel/logiciel posés par l'implantation de ces applications, de diminuer leur durée de conception et d'accroître leur qualité. La conception conjointe matériel/logiciel est l'une de ces méthodologies. Son but est de trouver le meilleur compromis entre implantation matérielle et implantation logicielle en gérant au mieux l'hétérogénéité de l'architecture cible dans le cadre d'un environnement de conception intégré qui couvre tout le cycle de conception de la spécification jusqu'au prototype de l'application à réaliser.

Nous nous attardons dans la suite de ce chapitre à présenter ce cycle de développement, à savoir : les définitions, le concept de modèle unifié, les caractéristiques souhaitables des outils de conception conjointe, la description et la comparaison de quelques environnements de conception conjointe matériel/logiciel....

2.2 La conception conjointe matériel/logiciel

L'approche de conception conjointe matériel/logiciel, encore nommée "conception concurrente", ou plus simplement "co-design" est donc née de la nécessité de développer des applications respectant des exigences de plus en plus sévères en performance, coût et délais, principalement lorsqu'il s'agit

de systèmes temps réel complexes (systèmes critiques embarqués par exemple). L'objectif d'une telle approche de conception conjointe est de produire un système matériel/logiciel optimal, qui répond aux spécifications données, en obéissant à l'ensemble de contraintes de conception d'ordre aussi bien technologique qu'économique (contraintes temps réel, de performance, de surface, de taille du code et de consommation...).

À la différence des approches traditionnelles, cette approche de conception conjointe permet le développement parallèle du matériel et du logiciel que l'on trouve dans les systèmes exigeant des architectures mixtes qui mélangent un ou plusieurs processeurs et un ou plusieurs circuits spécialisés qui communiquent (co-opèrent) entre eux. Cette façon de procéder permet de diminuer considérablement le temps de développement et également d'aboutir à de meilleures solutions par rapport aux approches traditionnelles [53] dans lesquelles on se focalise d'abord sur le matériel, en une seule itération, et ensuite on construit et modifie itérativement le logiciel de manière à ce que l'ensemble fonctionne correctement.

Il s'agit donc d'une nouvelle manière d'appréhender la conception de systèmes matériel/ logiciel en intégrant les techniques de conception du matériel et du logiciel dans un processus unique, structuré et automatique, de préférence à travers un environnement unifié qui préconise l'unification de la spécification pour englober à la fois la partie matérielle et la partie logicielle, sans présumer à l'avance de la manière dont seront implantées les différentes parties du système. Les avantages de cette intégration des techniques de développement du matériel et du logiciel sont l'accélération du processus de conception et la possibilité d'évaluer dynamiquement les différents compromis matériel/logiciel possibles.

Ainsi la conception conjointe peut être interprétée comme l'utilisation d'une méthodologie cohérente complète et efficace qui couvre les étapes de spécification, de validation et d'exploration des différentes alternatives d'implantation en matériel et en logiciel dans le but d'optimiser des critères de coût et/ou de performances.

Dans les prochaines années, ce type de conception va prendre une importance de plus en plus grande, étant donné la complexité des applications à traiter et surtout l'éventail des possibilités offertes par les technologies d'implantation. Les années à venir sont donc très prometteuses pour ce type de conception.

2.2.1 Motivation et avantages de la conception conjointe matériel/logiciel

Bien que l'approche de conception conjointe est une discipline émergente (la recherche dans ce domaine date d'environ 10 ans), de nombreux systèmes conçus par une approche co-design existent déjà et parmi les caractéristiques qui font du co-design une méthode de conception de plus en plus utilisée notons les suivantes [14] :

- **Flexibilité de la conception :** il est fréquent que la spécification subisse des modifications tout au long du processus de conception et de mise en oeuvre d'un système : raison par exemple de changements d'objectifs de l'utilisateur, d'une amélioration apportée, du développement de nouvelles versions, de compromis différents coût/performance ...

  Il est donc important que ces changements puissent être apportés sans avoir à refaire tout le travail déjà effectué. Étant basée sur une conception simultanée du matériel/logiciel, l'approche co-design permet la réalisation de systèmes où le choix d'implantation peut évoluer selon les besoins (certaines fonctions réalisées en logiciel au départ pouvant être intégrées au matériel par la suite), ce qui rend possible l'évolution des produits en fonction de la technologie (matérielle/logicielle) et offre plus de flexibilité. L'approche de co-design offre ainsi une vue plus intégrée du système

permettant de concevoir des produits suffisamment flexibles et rendant les éventuelles changements possibles et plus faciles à effectuer.

- **Délais de mise sur le marché réduits (*time to market*) :**
  le manque de concurrents favorise la mise sur le marché d'un produit. Il est donc économiquement important de lancer un nouveau produit le plus tôt possible, afin d'investir le marché avant l'apparition de concurrents. Le fait d'inclure la plus grande partie du système conçu par une approche co-design dans une méthodologie de conception unique et automatisée permet d'accélérer le processus de conception. De plus en autorisant la réutilisation, le co-design diminue considérablement le temps global séparant l'idée du produit final.
- **Exploration efficace de l'espace de conception :** le co-design permet d'explorer un plus grand nombre de solutions, contrairement aux approches traditionnelles de conception de systèmes digitaux où le travail sur chaque sous-système s'effectue en isolation totale. Le fait de considérer, en co-design, simultanément la conception des différents composants sous une méthodologie unique permet des compromis dynamiques entre matériel et logiciel au fur et à mesure que le processus de conception progresse. Une fois la majorité des choix effectués, il est donc possible de valider de nombreuses caractéristiques d'un système et cela bien avant la fin de conception.
- **Conceptions à moindre coût :** en facilitant l'exploration efficace de l'espace de conception, le co-design fournit un moyen pour l'analyse systématique des compromis en un temps raisonnable. Il est par conséquent fréquent que des alternatives plus intéressantes en coût soient trouvées par rapport aux méthodes de conception traditionnelles. Ce qui résulte en des produits meilleurs et moins chers. L'approche co-design permet donc d'aboutir à des solutions plus intéressantes en coût et en performances.
- **Prototypage rapide :** le développement de prototypes permet d'évaluer les propriétés du système à concevoir avant même que ce dernier ne soit réalisé. Le co-design permet de simplifier le développement de prototypes en raison de la vue globale du système qu'il autorise et de la capacité qu'il offre à produire rapidement des versions différentes du produit, et de tester le système de manière simple et rapide.

2.2.2 Domaines d'application de la conception conjointe matériel/logiciel

Le domaine d'application couvert par la conception conjointe est très vaste, puisqu'il comprend toutes les applications dont les impératifs économiques et les contraintes de conception nécessitent une architecture mixte. Le codesign est de plus en plus employé pour la conception-implantation de systèmes mixtes dans des domaines de plus en plus nombreux. Nous en citons, dans ce qui suit, quelques-uns des plus importants :

- **Les systèmes embarqués :** par système embarqué nous entendons un système digital qui représente un sous-système dans un système global et qui offre un service spécifique à ce système. Ces systèmes sont de bons candidats à la conception conjointe et leur utilisation classique consiste à contrôler d'autres systèmes (systèmes de contrôle d'un robot, de contrôle et de maintenance dans les automobiles, les avions, les procédés industriels,...). Ils opèrent généralement dans des environnements temps-réel.

- **Les télécommunications :** c'est l'un des marchés le plus en expansion actuellement (modems, téléphones mobiles...), notamment par le développement des télécommunications sans fil et d'internet. Il constitue l'une des plus intéressantes applications du codesign, car il nécessite la production à moindre coût de systèmes à contraintes strictes (performance, poids, consommation...) et ceci doit être fait dans un environnement de grande concurrence où un temps de mise sur le marché court est absolument crucial. Les produits sont nécessairement fournis en de nombreuses versions différentes mises à jour continuellement. Il est par conséquent important que les composants existants puissent être réutilisés aussi souvent que possible, d'où le besoin en produits flexibles conçus à faible coût.
- **Le traitement de signaux digitaux :** les systèmes conçus dans ce domaine effectuent des opérations de calcul intensif, souvent sous des contraintes de temps critiques. Le matériel est utilisé pour les calculs critiques en temps d'exécution et pour les communications externes, alors que le logiciel se charge des autres processus.
- **Les applications médicales :** le codesign est de plus en plus utilisé dans les applications médicales telles que le développement de prothèses orthopédiques intelligentes, les stimulateurs cardiaques, les prothèses auditives et autres systèmes digitaux.
- **Pour l'accélération d'algorithmes logiciels :** les systèmes matériel/logiciel ont pour but ici d'améliorer la performance de portions critiques d'applications décrites à l'aide d'un algorithme purement logiciel. Nous assistons, par exemple, actuellement à l'émergence d'une pléiade de nouveaux systèmes graphiques intégrés aux micro-ordinateurs (cartes de traitement graphique tridimensionnel, co-processeurs graphiques intégrés...).

2.2.3 Principales étapes de la conception conjointe matériel/logiciel

Comme dit précédemment, la conception conjointe est une approche qui intègre dans un même environnement la conception du logiciel et celle du matériel. Une approche typique de co-design comporte donc les étapes classiques du génie logiciel et de la micro-électronique (conception de circuits intégrés à très grande échelle). Il s'agit principalement de la spécification, la modélisation, le partitionnement, la synthèse du logiciel, la synthèse du matériel, la conception d'interface entre le logiciel et le matériel (i.e synthèse des communications) et la génération de code ou le prototypage. La succession de ces étapes forme le flot typique d'une approche de conception conjointe schématisé dans la figure 2.1 :

Le plus souvent la conception conjointe part d'une spécification conjointe unique décrivant l'architecture et/ou le comportement du système à concevoir. Après cette phase de spécification survient une phase de partitionnement du système ayant pour but de décomposer ce dernier en trois parties :

- Une partie matérielle implantée sous forme de circuits (FPGA, ASIC, ...).
- Une partie logicielle implantée sous forme d'un programme exécutable sur processeur, par exemple à usage général.
- Une interface de communication entre les deux parties.

Les partitions obtenues doivent ensuite être vérifiées et validées avant de passer à la phase de synthèse et d'implantation proprement dite. Des retours aux étapes précédentes (*feed-back*), plus précisément la phase de partitionnement, sont nécessaires tant qu'une solution "convenable" (c'est à dire répondant aux contraintes préalablement fixées), à défaut d'être optimale, n'a pas été obtenue.

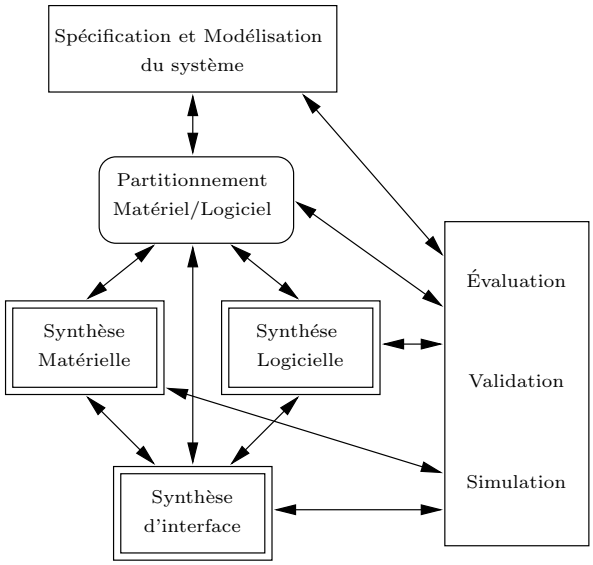


FIG. 2.1 – Flot de conception conjointe matériel/logiciel

1.2.3.1 Spécification et modélisation

C'est le point de départ du processus de conception conjointe, qui consiste, en général, à décrire les fonctionnalités du système à concevoir ainsi que toutes les contraintes qu'il doit satisfaire sans se soucier du découpage matériel/logiciel qui en suit. Le résultat de cette étape est une spécification fonctionnelle du système indépendante de tout détail d'implantation future, qu'elle soit matérielle ou logicielle. Cette spécification unifiée constitue donc le premier pas vers l'unification de la conception dans un processus de co-design. Il est donc indispensable que le langage de description utilisé ainsi que le modèle interne sous-jacent adopté pour représenter le système soient assez puissants pour inclure le logiciel, le matériel, ainsi que toutes les contraintes associées au système, on parle alors de "co-spécification".

En fonction du modèle interne adopté pour la représentation du système spécifié, on distingue dans la littérature différentes approches de conception conjointe [93, 55].

- La première classe correspond aux environnements de conception conjointe basés sur un modèle de spécification interne homogène (*homogeneous approach*), le système est décrit par une spécification unique et unifiée indépendamment de toute considération matérielle ou logicielle.
- La seconde classe correspond aux environnements de conception conjointe basés sur l'utilisation de modèles multiples pour spécifier les différentes parties du système. Chaque partie du système est spécifiée à l'aide d'un modèle correspondant à sa nature (*heterogeneous approach*).

Il est important de noter qu'une autre classification possible est basée sur le nombre de langages de spécification utilisés [56]. On distingue ainsi les approches homogènes (basées sur l'utilisation d'un langage de spécification unique) des approches hétérogènes (basées sur l'utilisation de langages spécifiques à chaque partie du système). Cependant, nous considérons qu'une telle classification est un peu trop superficielle, étant donné que souvent plusieurs langages de spécification utilisés au début du processus de spécification peuvent être basés sur le même modèle de représentation interne.

Enfin, il est intéressant de noter que le choix du formalisme (langage et modèle) de co-spécification utilisé est d'une importance primordiale, car il peut avoir une influence directe sur les autres étapes de conception, plus particulièrement le partitionnement. Ce choix du formalisme de co-spécification constitue aussi un premier pas vers l'unification de la conception.

1.2.3.2 Implantation

Nous appelons l'implantation la réalisation physique du matériel (par la synthèse) et du logiciel exécutable (par la compilation). Une caractéristique désirable des approches de co-design lors de l'implantation est qu'il est préférable qu'il n'y est pas un problème de continuité de modèle afin de préserver les propriétés formelles initiales. C'est à dire : les étapes successives de raffinement de la modélisation à la synthèse devraient toutes s'articuler autour du même modèle interne adopté en fixant à chaque étape certains détails d'implantation [57]. Les informations architecturales concernant les détails d'implantation seront prises en compte par le modèle lors de la phase d'implantation. Varea [58] appelle cela la fusion entre le modèle de spécification et la librairie de technologie.
Les étapes de cette phase d'implantation sont principalement :

- Le partitionnement matériel/logiciel
- La synthèse du code pour la partie matérielle et la partie logicielle (co-synthèse)

1.2.3.2.1 Partitionnement matériel/logiciel Après la phase de spécification survient une phase de partitionnement ayant pour but d'assurer la transformation des spécifications du système en une architecture composée d'une partie matérielle et d'une partie logicielle. Cette transformation s'effectue habituellement en deux phases : la sélection d'une architecture multicomposant constituée souvent de composants matériels (ASICs, FPGAs, composants standards...) et de composants logiciels (ASIP, DSP...), et l'allocation des éléments du modèle fonctionnel spécifiant l'application sur les composants de cette architecture. Cette phase de partitionnement consiste donc à déterminer les parties du système qui seront réalisées en matériel et celles qui seront réalisées en logiciel ainsi que l'interface entre ces différentes parties. La définition de ces différentes partitions est souvent guidée par l'ensemble des contraintes définies initialement par le concepteur (espace réservé au logiciel et au matériel, temps d'exécution, taux de parallélisme, taux de communication, consommation, temps de conception et coût industriel, poids,...). En effet, une fonctionnalité donnée peut avoir diverses implantations (en matériel ou en logiciel), chacune avec ses propres caractéristiques de coût et performance. Il est donc nécessaire d'explorer différentes possibilités d'implantation en matériel et en logiciel avant de se décider sur la partition du système qui satisfait de façon optimale les contraintes de coût et/ou de performance imposées. Le partitionnement est donc d'une importance critique car il a un impact important sur les caractéristiques de coût/performances du système final.

Les nombreuses méthodes adoptées dans la littérature pour résoudre cette problématique de partitionnement logiciel/matériel varient toutes sur l'approche utilisée. Nous pouvons tout de même les classer comme il est montré sur la figure 2.2 en fonction d'un certain nombre de facteurs tels que le degré de granularité, le fait que le partitionnement soit manuel ou automatique....
Le degré de granularité indique en fait le niveau de finesse des éléments du modèle fonctionnel adopté lors du processus de partitionnement. Ce dernier varie souvent du grain fin "*fine -grained*" (niveau opérations, instructions), au grain moyen "*medium-grained*" (niveau macro-opérations, blocs de base...),

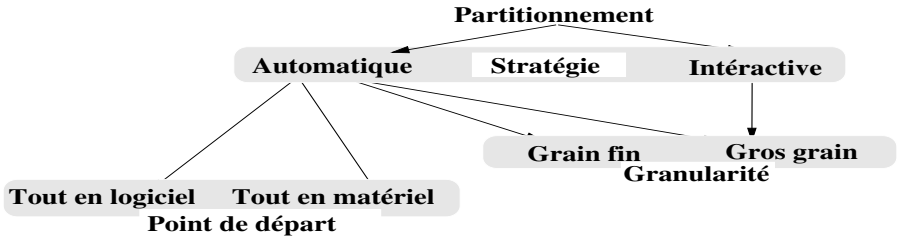


FIG. 2.2 – Approches de partitionnement

au gros grain "*coarse-grained*" (niveau processus, tâches, fonctions,...). Il n'existe cependant pas de consensus quant au niveau à adopter pour le partitionnement, car chacun présente des avantages et des inconvénients. Toutefois, les approches interactives adoptent souvent une granularité de haut niveau afin de pouvoir traiter de manière efficace des systèmes de grandes tailles pour lesquels il est de plus en plus difficile de gérer manuellement un partitionnement à grain fin. Les approches automatiques basées sur des heuristiques guidées par leurs fonctions de coût quant à elles adoptent souvent une granularité plus fine.

1.2.3.2.2 Co-synthèse Une fois le partitionnement matériel/logiciel terminé les décisions liées à l'implantation souhaitée sont fixées, l'étape de co-synthèse (étape d'implantation proprement dite) se chargera de transformer les spécifications fonctionnelles en descriptions directement implantables sur les composants matériels et logiciels de l'architecture cible. Son rôle est d'assurer la conception physique de la partie matérielle (par synthèse de haut niveau), la génération du code exécutable correspondant à l'implantation de la partie logicielle (par la compilation) ainsi que la synthèse des interfaces matériel/logiciel en vue de produire finalement selon le cas un prototype physique du système conçu ou le produit fini lui même. Pour ce faire, le processus de co-synthèse doit pouvoir s'interfacer et se connecter aux outils de développement logiciel et matériel existants supportés par les composants constituant l'architecture d'exécution du système conçu. Cet interfaçage est rendu possible via la mise en oeuvre d'une génération automatique de code source accepté comme entrée par ces outils.

Synthèse du logiciel La synthèse logicielle consiste en la génération d'un code exécutable correct et efficace correspondant à l'implantation de la partie logicielle à partir d'une spécification de haut niveau. Les intérêts de cette synthèse logicielle résident dans la diminution du coût de développement et surtout l'augmentation de la fiabilité du code généré. La complexité de cette étape dépend du type de processeur utilisé. On distingue souvent la synthèse du logiciel pour les processeurs généraux (Pentium, SPARC,...), de la synthèse pour les processeurs de traitement du signal et de la synthèse pour les ASIP (*Application-Specific Instruction set-Processsors*).

Le plus souvent, une méthode efficace de synthèse consiste à définir un schéma d'implantation logicielle sans utiliser d'exécutifs temps-réel. Une autre méthode consiste à utiliser un exécutif temps-réel. Une solution intermédiaire existe qui utilise judicieusement les qualités de ces deux méthodes.

Synthèse du matériel La synthèse matérielle consiste à transformer la spécification de haut niveau de l'application vers un circuit électrique. On distingue généralement deux niveaux de synthèse

matérielle : synthèse logique et synthèse comportementale. La synthèse logique consiste en la transformation d'une description de niveau RTL (*Register Transfer Level*) en un réseau de portes logiques interconnectées qui réalise les fonctionnalités souhaitées. La synthèse comportementale consiste quant à elle en la transformation d'une description comportementale faite en langage de haut niveau (un algorithme) vers une architecture décrite au niveau RTL, composée d'une partie chemin de données et d'une partie chemin de contrôle. Du fait que lors de la co-synthèse la synthèse du matériel et du logiciel sont traitées globalement de façon identique et à des niveaux d'abstraction similaires, la synthèse du matériel dans le cadre de co-design repose sur la synthèse au niveau comportemental (i.e synthèse d'architecture au niveau RTL). Les outils classiques de CAO permettent par la suite le passage quasi automatique du modèle RTL ainsi généré à la puce réalisant ainsi la synthèse logique. Les intérêts de cette synthèse automatique du matériel résident dans l'augmentation de la productivité du matériel, la diminution du coût de développement et surtout l'augmentation de la fiabilité du circuit généré.

Synthèse d'interface ou synthèse des communications La synthèse des communications est la réalisation des interfaces de communications entre les différentes ressources. Plus précisément, elle permet de définir les protocoles de communications et les interfaces d'E/S entre les différentes partitions (sous-systèmes communicants issus du partitionnement). En effet, les différentes parties du système échangent des informations à travers des mécanismes dédiés à la communication. Ces mécanismes peuvent être mis en ouvres en matériel (bus, contrôleurs...) ou en logiciel (pilotes, protocoles,...). Certains concepteurs préconisent l'utilisation de protocoles "standard" ou tout au moins connus, alors que d'autres développent leurs propres protocoles.

1.2.3.3 Validation

À ce stade, l'activité de validation permet de vérifier que les sous-systèmes obtenus après les étapes successives de partitionnement et de synthèse réalisent lorsqu'ils fonctionnent ensemble les comportements attendus du système global et que l'implantation obtenue préserve bien les propriétés du système spécifié.

Signalons tout de même qu'une telle activité de validation est transversale à tout le processus de co-design. À la fin de chaque étape du processus, il est nécessaire d'évaluer si les objectifs de conception ont été atteints pour pouvoir procéder sans risque à la prochaine étape. Cette validation permet donc d'évaluer si les critères nécessaires pour passer à la prochaine étape sont satisfaits ou pas. Si ce n'est pas le cas, le concepteur doit ré-itérer l'étape actuelle jusqu'à ce qu'il obtient une validation réussie. Les activités de validation à la fin de chaque étape bien qu'elles ont toutes les mêmes objectifs (i.e vérifier si la conception est acceptable), elles diffèrent néanmoins sur le type d'approche, les outils etc. Le fait d'avoir une activité de validation à la fin de chaque étape assure l'homogénéité de la conception vis à vis du respect des objectifs de l'ensemble des étapes précédentes de conception.
L'activité de validation peut être effectuée selon trois approches différentes :

- La vérification (ou plus précisément la co-vérification) formelle par l'application des méthodes de preuves formelles à différents niveaux d'abstraction permettant de vérifier si les cospécifications sont complètes et correctes et de vérifier la conformité entre ces cospécifications et l'implantation finale,
- La simulation par l'intermédiaire d'outil de co-simulation permettant la simulation conjointe de

la partie logicielle, de la partie matérielle et de l'interface entre les deux,

– Le prototypage par la génération rapide d'une implantation prototype permettant d'évaluer, de valider la faisabilité du système ainsi que ses caractéristiques.

Pour conclure cette description des différentes phases du processus de conception conjointe, il est intéressant de noter que contrairement aux premières approches, dans lesquelles la recherche en co-design avait pour objectif la transformation automatique d'une spécification en une implémentation, les chercheurs tendent actuellement à mettre l'accent sur des problèmes plus spécifiques liés au processus de conception, en particulier, la spécification, la modélisation architecturale, le partitionnement, l'estimation des caractéristiques, les heuristiques d'exploration, la synthèse de la communication, la validation, et la génération automatique de code.

2.3 Systèmes de conception conjointe matériel/logiciel

Le succès que connaît actuellement le co-design a suscité l'intérêt des nombreux groupes de recherche tant dans le monde universitaire que dans le monde industriel regroupant ainsi des professionnels de conception de circuits, de conception de logiciel et de spécification de systèmes. Ces efforts ont donné lieu à de nombreux environnements avec des objectifs et des approches différentes.

Dans ce qui suit nous présentons quelques-uns des nombreux environnements de conception conjointe développés. Notre choix des environnements présentés a porté principalement sur des systèmes complets qui proposent des méthodologies et des outils logiciels supportant la méthodologie proposée. Nous nous sommes aussi attardés à présenter des systèmes dédiés à des domaines d'applications variés et adoptant des approches différentes afin de couvrir un large spectre des travaux effectués dans le domaine de la conception conjointe :

2.3.1 Ptolemy/PtolemyII

PtolemyII [61][62] est le successeur de Ptolemy 0.7 (appelé aussi "Ptolemy Classic" [63][65]) dont il reprend de nombreuses caractéristiques. Cet outil de recherche, développé à l'université de Californie, à Berkeley, sous la direction du Prof. Edward A. Lee, propose un environnement hétérogène de modélisation, de simulation et de construction d'applications concurrentes, dédié aux systèmes embarqués réactifs. Par hétérogénéité, on entend ici l'hétérogénéité des modèles de calcul, et non l'hétérogénéité logicielle ou matérielle.

Cet environnement de conception conjointe vise la conception des systèmes embarqués, spécifiquement les systèmes qui combinent plusieurs technologies différentes y compris, par exemple, composants analogiques/digitaux, matériels/logiciels, et dispositifs mécaniques/électroniques (MEMS–*Systèmes Micro-Electro-Mécaniques*) et qui combinent également plusieurs types de traitements, comme par exemple le traitement du signal, le contrôle, les automates de décision séquentielle, ou les interfaces utilisateur.

Cette variété des domaines d'application visés par le projet Ptolemy a nécessité le support d'une variété de formalismes de spécification, chacun bien adapté à un domaine spécifique. Pour cela, Ptolemy supporte la construction et l'interopérabilité de modèles exécutables selon des modèles de calculs très variés en autorisant l'imbrication hiérarchique des modèles. La décomposition récursive d'un système

complexe en sous-systèmes réalisées par des modèles de calcul peut être hétérogène, c'est-à-dire que chaque décomposition peut avoir un modèle de calcul différent.

Pour manipuler la diversité des modèles de calcul des différents sous-systèmes, Ptolemy se base sur la notion de "domaine" (*"domains"*), où chaque domaine est un objet encapsulant un modèle de calcul et possédant sa propre sémantique. Néanmoins, tous les domaines utilisent le modèle flot de données comme modèle de base. Une application sous Ptolemy est donc décrite sous la forme d'un graphe d'acteurs (noeud d'exécution) reliés par des arcs où l'interprétation des acteurs et des arcs diffèrent selon le "'domaine"' choisi. Ptolemy propose un certain nombre de domaines dont les plus importants sont : le modèle à événements discrets (DE*"Discrete-Events"*), le modèle à base de machines d'états finis (FSM *Finite-State-Machine*), le modèle à base de processus communicants (CSP *Communicating Sequential Processes*), le modèle à base de réseaux de processus synchrone (SDF *Synchronous Data Flow*) etc.

Malgré la diversité des domaines offerte par Ptolemy, il offre cependant une infrastructure de spécification unifiée pour la partie matérielle et la partie logicielle sous forme de modèles de calcul ce qui facilite la migration de fonctions entre les deux types d'implantation.

Cette étape de spécification est suivie par une étape de partitionnement mas telles que le coût des communications, la surface ou la vitesse. Lors de la synthèse, le code généré pour la partie logicielle pourrait être du C ou de l'assembleur selon le processeur cible et du code SILAGE [66] pour la partie matérielle. La simulation du système hétérogène a lieu une fois que la synthèse du logiciel, du matériel et des interfaces sont faites. Les résultats de cette simulation permettent de vérifier si la conception est conforme à la spécification initiale. Cependant, à l'issue des étapes de synthèse du matériel et du logiciel, quelques estimations de performances sont faites afin de valider la conception. Les estimations concernent la surface, le chemin critique et l'utilisation de bus et de composants. L'architecture cible, dans Ptolemy, comprend une variété de processeurs qui peuvent avoir différentes configurations (mono-processeur, multiprocesseurs, architectures parallèles, etc.). Notons néanmoins que la spécification de cette architecture est entièrement à la charge de l'utilisateur, qui doit la simuler soit explicitement soit implicitement dans ses modèles.

Il est bon de noter que dans l'environnement Ptolemy, l'accent est mis surtout sur la simulation des systèmes. En effet, bien que Ptolemy soit un environnement fiable de simulation et de modélisation de systèmes complexes très hétérogènes à plusieurs niveaux de raffinements successifs, ses capacités de génération de code d'une implantation exécutable en temps réel des systèmes cibles restent limités. Ptolemy peut générer du code C dont la qualité n'est souvent pas satisfaisante mais il peut rarement générer un code VHDL synthétisable, ce dernier n'est absolument pas optimisé. Le principal travail de fond du projet Ptolemy est la définition d'un cadre conceptuel pour la comparaison des différents modèles de calcul. Plusieurs projets ont repris les travaux de Ptolemy pour construire des outils de cosimulation. Ainsi l'environnement de conception logicielle-matérielle Polis [68] utilise le domaine DE de Ptolemy comme support de réalisation des instances de cosimulation pour la validation au niveau RTL des systèmes logiciels-matériels. SPW [67] de Cadence utilise les concepts de Ptolemy pour la validation par cosimulation des systèmes orientés traitement de signal.

2.3.2 Cosyma

Cosyma (*COSYnthesis for eMbedded Architectures*) [69, 70] est un environnement de conception conjointe pour les microcontrôleurs et les systèmes embarqués dominés par le contrôle, développé à l'Université de Braunschweig, en Allemagne, sous la direction du Prof. Rolf Ernst. La description du système en entrée se fait par l'intermédiaire de processus communicants (CSP), en utilisant des fonctions en langage C^x. Ce langage correspond à une extension minimale du langage C afin de permettre le traitement de processus parallèles et d'inclure des contraintes de temps. Cette description en C^x est strictement séparée des directives de contraintes et d'implantation, pour minimiser les extensions au langage C et pour permettre plus de portabilité. Elle a pour but d'utiliser une spécification homogène unifiée permettant de migrer une fonctionnalité donnée entre logiciel et matériel. La description d'entrée donnée en C^x est traduite par la suite en une représentation interne sous forme de graphe appelé graphe syntaxique étendu "ES Graph" (*Extended syntax graph*) en utilisant une version adaptée du compilateur *Stanford SUIF* [72]. Un ES Graph est un modèle de graphe similaire au graphe flots de données et de contrôle (CDFG control data flow graph), qui décrit une séquence de déclarations, définitions et d'instructions en intégrant un graphe de flot de données contenant des informations sur les dépendances des données. À la suite de cette traduction du C^x en ES graph, une simulation préliminaire et un profilage (*profiling*) sont réalisés pour extraire les informations nécessaires au partitionnement. Le partitionnement matériel/logiciel est automatique et il est basé sur un algorithme de recuit simulé (*simulated annealing*). La stratégie, adaptée pour le partitionnement matériel/logiciel en Cosyma, essaye d'extraire de façon itérative, à partir d'une solution purement logicielle, les parties critiques pour les implanter en matériel en exploitant les informations obtenues par le profilage, jusqu'au respect des contraintes temporelles. Le partitionnement dans Cosyma cherche ainsi à satisfaire les contraintes temps réel tout en minimisant les coûts matériels et le temps de réponse du système de CAO. L'estimation des coûts de communication et l'optimisation du code sont effectuées à partir d'une technique basée sur les algorithmes d'Aho, Sethi et Ullman [71] pour l'analyse de flots de données. Le résultat de cette analyse des flots de données peut être utilisé pour guider des optimisations telles que la propagation de constantes, l'optimisation arithmétique, l'élimination de sous-expressions communes. Suite au partitionnement matériel/logiciel, les parties à implanter en logiciel sont traduites en langage C et les parties à implanter en matériel sont traduites en langage HardwareC (autre extension du langage C), qui peut être synthétisé par *Olympus* [73, 74], un outil de conception de circuits numériques développée à l'Université de Stanford. Pour la synthèse de haut niveau (HLS–*High-Level Synthesis*), Cosyma utilise le *Braunschweig Synthesis System* (BSS), qui est un système de synthèse de haut niveau développé spécifiquement pour la conception de coprocesseurs rapides. Un outil de synthèse au niveau RTL, le *Synopsis Design Compiler*, génère la *netlist* finale. Pour la synthèse logicielle, un compilateur C standard est utilisé. L'environnement Cosyma est limité généralement à une architecture figée composée d'un processeur RISC (*Reduced Instruction Set Code*) standard avec un coprocesseur spécifique généré automatiquement (circuit intégré (ASIC)), une RAM (*Random Access Memory*) rapide et un bus. Enfin, le fait que Cosyma soit basé sur une migration sélective des parties matérielles en vue du respect des contraintes temporelles limite relativement l'exploitation potentielle des composants matériel. De plus, l'hypothèse que les parties matérielles et logicielles s'exécutent de manière entrelacée (et non pas concurrente) résulte en un système qui sous-utilise ses ressources. Les autres inconvénients de Cosyma sont le manque de précision dans l'estimation des coûts ainsi que les temps élevés de compilation.

2.3.3 Vulcan

Vulcan [75] est un environnement de conception conjointe, développé à l'Université de Stanford, principalement dédié aux systèmes temps-réel. La description initiale en entrée de cet environnement est une description fonctionnelle donnée en HardwareC. Il s'agit d'une extension du langage C permettant d'inclure le parallélisme et la description des aspects matériels. Cette description en HardwareC consiste en un ensemble de processus interagissants qui sont instanciés dans des blocs. Un processus s'exécute en parallèle avec d'autres processus de la spécification et se ré-initialise automatiquement à sa terminaison. L'approche de conception adoptée par Vulcan est basée principalement sur l'utilisation de contraintes temporelles. Elle est typiquement orientée vers l'architecture et suit les étapes suivantes : saisie de la spécification, partitionnement matériel/logiciel et synthèse du modèle partitionné en composants matériels et logiciels interconnectés dans une architecture cible.

L'algorithme de partitionnement commence avec une partition initiale où toutes les opérations, exceptées celles à délai non déterminé sont affectées au matériel (par exemple : les opérations non déterministes relatives aux opérations de boucles dépendant des données qui définissent le début des processus sont considérées comme des opérations dont le délai n'est pas déterminé). Contrairement à Cosyma, la partition est raffinée par la migration d'opérations du matériel vers le logiciel afin d'obtenir une partition à moindre coût tout en respectant les contraintes temporelles. L'approche de Vulcan utilise un ensemble de modèles de graphes de séquencement avec des contraintes temporelles entre les opérations. L'application spécifiée est ainsi transformée en un ensemble de modèles de graphes de séquencement où chaque processus est compilé en un graphe de séquencement (*flow graph model*). Un ensemble de modèles de graphes de séquencement est implanté en logiciel et un autre est implanté en matériel. La partie logicielle de l'application est découpée en un ensemble de threads de telle façon qu'une thread est définie comme un ensemble d'opérations linéarisées (basic block). Chaque graphe devient une thread et suivant les opérations dont le délai n'est pas borné, plusieurs threads peuvent être créés à partir d'un graphe. Les threads sont construites de façon à ce que cet ordre partiel soit respecté. L'exécution concurrente des threads est effectuée à travers un modèle d'exécution supportant l'entrelacement. L'interface matériel/logiciel consiste en des files de données et un contrôleur qui maintient les identificateurs pour les processus activés dans l'ordre de l'arrivée de leurs données. Enfin, l'architecture ciblée par l'environnement Vulcan utilise un seul processeur à usage général qui est intégré avec un seul circuit intégré (ASIC).

Bien que Vulcan offre un environnement de conception complet qui supporte la spécification, le partitionnement automatique, et la co-synthèse des parties logicielles et matérielles, il reste tout de même limité à un niveau de spécification de bas niveau basé sur HardwareC. De plus l'architecture qu'il cible est assez restreinte.

2.3.4 Polis

L'environnement *Polis* [76, 77], développé à l'Université de Californie, à Berkeley, est un environnement qui implante une méthodologie de spécification, de synthèse automatique et de validation de systèmes réactifs temps-réel à contrôle intensif de taille réduite tels ceux largement utilisés dans l'industrie automatique.

La conception du système est réalisée à partir d'une représentation unifiée du comportement du système capable de spécifier à la fois les aspects matériels et logiciels. Ce format unifié est conservé tout

au long du processus de conception, afin de préserver les propriétés formelles de la spécification.

Le format unifié utilisé par Polis est basé sur une représentation de type machine à état fini, appelée machine d'états finis pour la co-conception (CFSM *Codesign Finite State Machine*) [76]. La différence entre les deux modèles est due au fait que dans une CFSM la communication synchrone d'une FSM est remplacée par un modèle de communication asynchrone à temps de réaction illimité et à base d'événements. Chaque élément d'un réseau de CFSM, décrit un composant du système à modéliser. Dans Polis, l'utilisateur peut spécifier son application en langages de haut niveau tels que Esterel, sous-ensemble de Verilog ou VHDL et FSM graphiques. Ces langages sont directement traduits en CFSM. Polis inclut un traducteur du formalisme CFSM vers le formalisme FSM qui peut être utilisé directement pour la vérification du système (projet VIS " *Verification Interacting with Synthesis*" [78]). Il est bon de noter qu'une spécification CFSM réussit à représenter une spécification indépendante de l'implantation qu'elle soit matérielle ou logicielle, ce qui permet de retarder la décision des choix d'implantation et offre de nombreuses possibilités d'exploration de l'espace des solutions d'implantation. Les CFSMs offrent aussi un modèle synthétisable et vérifiable à cause du grand nombre de théories et outils développés pour le modèle FSM qui peuvent être facilement adaptés aux CFSMs. Cependant le modèle de communication supporté par les CFSMs reste limité et insuffisant pour représenter les systèmes de traitements intensifs de données, c'est pourquoi Polis est principalement utilisé pour les systèmes embarqués à contrôle intensif. Le flot de conception de *Polis* est décrit comme suit : initialement la spécification est validée par co-simulation logicielle/matérielle au niveau du système. Actuellement, Polis utilise l'environnement *Ptolemy II* pour réaliser cette co-simulation. Mais, il est possible d'utiliser n'importe quel outil de co-simulation car le code VHDL généré en sortie contient toutes les informations nécessaires à la co-simulation. Cette étape préliminaire de co-simulation permet de guider les décisions du concepteur lors du partitionnement matériel/logiciel, de la sélection du CPU–*Central Processor Unit* et de l'ordonnancement. Ces tâches sont basées sur l'expérience du concepteur, et donc très difficiles à automatiser. Après partitionnement du système, la synthèse du matériel est réalisée selon les techniques de synthèse logique fournies par SIS [78]. Chaque CFSM, interprétée comme une spécification RTL, peut être traduite (*mapped*) en BLIF (*Berkeley Logic Interchange Format*), XNF (*Xilinx Netlist Format*), VHDL ou Verilog). La synthèse du logiciel traduit la CFSM en une structure logicielle composée d'une procédure pour chaque CFSM et d'un système d'exploitation temps réel simple. Le comportement réactif est synthétisé à travers un processus de deux étapes : (1) le comportement souhaité est implanté et optimisé en utilisant une représentation de haut niveau, indépendante du processeur, similaire à un graphe de flots de contrôle et de données (CDFG *Control-Data Flow Graph*), (2) le CDFG est traduit vers du code C portable et utilise n'importe quel compilateur pour l'implanter et l'optimiser sur un micro-contrôleur. Un prédicteur temporel analyse le logiciel et estime la taille du code et les temps d'exécution. Finalement, les interfaces entre les différentes implantations (matériel/logiciel) sont automatiquement synthétisées. L'architecture ciblée par Polis est composée de microcontrôleur et d'un matériel spécialisé. Polis dispose ainsi que de deux partitions : une matérielle et une logicielle et n'offre pas des techniques d'estimation de performance pour des modèles de processeurs plus complexes que le microcontrôleur simple supporté [50]. Un outil de conception conjointe de systèmes au niveau système (VCC Virtual Component Co-design) basé sur les concepts de Polis est commercialisé par Cadence [51, 80].

2.3.5 Cosmos

Le projet COSMOS développé par l'équipe TIMA (*Techniques of Informatics and Microelectronics for computer Architecture*) en France est à la fois un outil(s) et une méthodologie de conception conjointe semi-automatique orientée vers la conception de systèmes distribués [82]. Le domaine d'application visé est celui des systèmes hétérogènes communicants en particulier celui des télécommunications. L'environnement Cosmos accepte comme entrée le langage SDL (*Spécification and Description Language*, standard CCITT) et produit un modèle distribué en C/VHDL. La spécification initiale en SDL, vue comme un ensemble de processus concurrents, est traduite dans un format intermédiaire d'unification de spécification appelé Solar sur lequel seront effectuées toutes les étapes ultérieures de raffinement successives. Ce modèle combine deux concepts puissants : Les machines d'états finis étendues (EFSM) pour la description du comportement et l'appel de procédure à distance (RPC Remote Procedure Call) pour la spécification de la communication de haut niveau. Un système en Solar est donc décrit comme étant une hiérarchie d'unités de conception communicantes.

Cosmos utilise une approche transformationnelle pour le raffinement d'une spécification en un modèle C/VHDL. Chaque étape de conception réduit le fossé entre la spécification et la réalisation en fixant certains détails d'implantation ou en préparant d'autres transformations. La première étape est celle de la saisie des spécifications des différentes parties du système décrites chacune dans le langage le mieux adapté. Une fois simulée, ces spécifications seront unifiées par traduction dans le format intermédiaire appelé Solar. Ensuite, les étapes de synthèse et de transformation peuvent commencer. La première étape de cette synthèse est le partitionnement matériel/logiciel qui repose sur le découpage et la fusion de machines d'états finis étendues. L'approche adoptée est interactive à travers l'utilisation d'un outil appelé Partif [83] et n'est donc plus entièrement automatique, néanmoins un certain nombre d'heuristiques ont été proposées. La seconde étape est la synthèse des communications qui consiste à transformer le système composé de processus communicants à travers des canaux en un ensemble de processeurs interconnectés communicants à travers des bus et des signaux et partageant le contrôle de la communication. L'étape suivante est celle du prototypage virtuel qui permet de générer un code exécutable C/VHDL pour chaque processeur abstrait résultant du partitionnement. Enfin, l'étape finale de ce processus de synthèse est la génération d'architecture qui implante la spécification initiale. Cette opération est réalisée en utilisant des compilateurs standards pour les parties logicielles (code C) et des outils de synthèse architecturale ou logique pour les parties matérielles (code VHDL). L'architecture peut être un circuit intégrant plusieurs processeurs programmables ou câblés ou un réseau de processeurs. Actuellement, dans le projet Cosmos, les FPGAs sont utilisés pour réaliser les parties matérielles. Notons en particulier que dans la méthodologie développée par le projet Cosmos, l'accent est mis principalement sur la synthèse des interfaces de communication. Cette méthodologie peut être vue comme un processus de compilation guidé par l'utilisateur humain où le concepteur fournit un effort supplémentaire pour produire une implantation efficace.

2.3.6 CoWare

CoWare est un environnement pour la conception de systèmes hétérogènes matériel/ logiciel, développé à l'IMEC (Belgique) [84]. Il est orienté vers la conception des systèmes distribués et couvre plusieurs domaines d'applications, en particulier les systèmes de télé- communications embarqués. Pour spécifier l'application cible, CoWare utilise un modèle de données basé sur des processus communicants

hétérogènes. Ce modèle permet de séparer les descriptions des comportements fonctionnels et de communication. La même méthode de spécification est utilisée pour modéliser l'architecture cible. L'objectif est d'offrir ainsi une représentation unifiée du système spécifié permettant la conception du matériel en parallèle avec le développement du logiciel. Dans CoWare, l'application est spécifiée par l'intermédiaire de processus communicants dont les comportements peuvent être décrits par des langages comme C, C++, DFL (*Data Flow Language*) ou VHDL. À partir de cette spécification fonctionnelle, des distributions et des partitionnements matériel/logiciel différents peuvent être explorés par l'utilisateur de façon interactive. L'implantation de l'application est effectuée automatiquement par l'outil de synthèse *Symphony*. Ensuite, les composants matériels et logiciels générés par *Symphony* [85] peuvent être implantés par d'autres outils de CAO commerciaux, comme le compilateur C *ARM*, le compilateur VHDL *Synopsis* et l'environnement *Cathedral*. Pendant le processus de conception, CoWare permet de réaliser la co-simulation de spécifications à différents niveaux d'abstraction assurant ainsi une validation continue tout au long de la conception. Pour cela, les simulateurs *Synopsis* pour le code VHDL et *ARM* pour le code C ont été intégrés dans son environnement. CoWare cible des architectures composées de plusieurs processeurs logiciels programmables de type DSP (Digital Signal Processor) ou micro-contrôleurs, et des processeurs matériels dédiés.

Notons que dans l'environnement CoWare, l'exploration et la validation des solutions d'implantation d'un système sont extrêmement rapides car le matériel est spécifié à un niveau d'abstraction très haut ce qui donne une augmentation de la vitesse de simulation et de modélisation. Compte tenu que le principal objectif de CoWare est l'intégration et la manipulation de la communication dans les systèmes embarqués, les interfaces sont décrites au niveau de l'application, permettant de ce fait une grande modularité et un niveau d'abstraction élevé, capable de manipuler une grande classe de modèles de communication. Cependant, l'utilisateur de CoWare est prié de fournir des bibliothèques de communication grandes et sophistiquées, ce qui est tout à fait difficile à concevoir et à mettre au point.

Enfin, ce système développé initialement comme outil de recherche à l'IMEC a été industrialisé par la suite par CoWare Inc.

2.3.7 GrapeII

GRAPE II [86, 88](*Graphical RApid Prototyping Environment*), développé à l'Université Catholique de Leuven, en Belgique, est un environnement de conception conjointe pour l'émulation temps-réel de systèmes synchrones à base de DSP. GRAPE II permet la spécification, l'estimation de ressources, le *retiming*, le partitionnement, la distribution, l'ordonnancement, le routage, la génération de code, la compilation, le débogage, la simulation et l'émulation des applications synchrones multi-vitesse (*multi-rate*) sur des architectures-cible hétérogènes, constituées par de multiples processeurs DSP et des FPGAs.

Dans cet environnement, l'application est spécifiée d'une façon totalement indépendante de l'architecture cible qui va l'implanter : le concepteur peut utiliser son éditeur graphique hiérarchique pour dessiner le graphe de dépendances de données ou il peut se servir de l'interface pour l'environnement *DSP Station* de *Frontier Design*, qui lui permet de spécifier l'application dans le langage flot de données DFL (dérivé du langage *Silage* [66]). L'architecture est spécifiée de façon similaire, tout en utilisant l'éditeur graphique. On peut spécifier la fréquence d'horloge, le taux et le protocole de communication, la quantité de mémoire disponible, etc. Une fois l'architecture définie, GRAPE II estime la quantité de ressources nécessaires à l'exécution de chaque tâche de l'application. Pour des cibles microprocesseurs,

il estime la latence, la taille du code et de la mémoire, et des ressources dédiées comme les convertisseurs analogique-numérique. Pour des cibles FPGA, il estime les ressources en termes du nombre de blocs logiques configurables (CLB–*Configurable Logic Block*), d'entrées/sorties, etc. À partir de cette estimation de ressources des tâches de calcul de l'application et des ressources disponibles décrits dans la spécification de l'architecture matérielle cible, GRAPEII effectue le partitionnement automatique, l'affectation, le routage et l'ordonnancement pour des cibles hétérogènes en utilisant des heuristiques de type *"branch and bound"*. Finalement, la dernière phase produit du code C ou VHDL pour chacun des composants de l'architecture mixte. GRAPE II génère automatiquement du code VHDL pour les outils de synthèse matérielle et du code C ou assembleur pour les compilateurs DSP. Notons que pour assurer une mise en correspondance efficace de l'algorithme de l'application sur l'architecture cible, GRAPEII effectue une exploration de l'espace de conception basée sur des transformations de déroulage (*"folding"*, *"unfolding"*, *"retiming"*, *"clustering"*, *etc*) appliquées au graphe algorithmique de l'application.

Enfin, GRAPE II est un environnement ouvert. Le concepteur peut ainsi y ajouter ses propres outils. GRAPE II est commercialisé sous le nom *Virtuoso Synchro* par *Intelligent Systems International*.

2.4 Caractéristiques désirables des systèmes de conception conjointe

Après avoir passé en revue quelques environnements représentatifs du domaine de la conception conjointe, nous avons jugé indispensable de rappeler ici les principales exigences et caractéristiques que doit posséder un système de conception conjointe, avant de dresser le bilan comparatif des ces différents systèmes. Une telle liste de caractéristiques pourrait servir à indiquer les défis actuels ainsi que les domaines qui posent toujours des problématiques et méritent une attention particulière des efforts des futurs travaux de recherche. Ces caractéristiques portent sur plusieurs aspects du processus de la conception conjointe :

- **Approche orientée modèle :**
  Pour s'affranchir de la diversité et de l'évolution constante des langages de spécification, il est de plus en plus souhaitable, voir indispensable, d'adopter une approche de conception basée sur l'utilisation d'un modèle de représentation interne autour duquel s'articuleront les différentes activités du processus de conception conjointe. Ainsi, tous les traitements des différentes étapes du processus de raffinement de la spécification à l'implantation, seront effectués sur le modèle interne décrivant le système à concevoir. Une telle approche indépendante des langages de spécification s'avère être de plus en plus une solution attractive pour la conception conjointe des systèmes mixtes matériel/logiciel.
  Notons cependant que certains concepteurs continuent même à confondre langage de spécification et modèle interne et la majorité ne voit donc pas la nécessité d'avoir un modèle interne à leur environnement de co-design. Toutes les opérations sur le code décrivant le système à concevoir sont effectuées dans le langage de co-spécification choisi [89, 90], ce qui limite considérablement les environnements de conception conjointe qu'ils proposent. Edwards [91] affirme que la distinction entre le langage et le modèle de calcul sous-jacent est important et Stoy [14] considère que le fait que des approches de co-design n'incluant pas de modèle de représentation défini formellement signifie qu'elles sont construites sur des bases très instables.

La définition et le choix judicieux d'un modèle de représentation interne pour un système de conception conjointe est donc d'une importance primordiale, car il peut avoir une influence directe sur le déroulement des différentes étapes du processus de conception et par conséquent sur les performances et la qualité du système conçu. Ce choix est souvent guidé par plusieurs considérations :

Tout d'abord, un processus de conception est souvent perçu comme une séquence de tâches de raffinement caractérisées par une transformation d'une spécification plus abstraite en une spécification plus détaillée. Il est donc désirable de disposer d'un modèle de spécification permettant de passer aisément d'une spécification à une autre sans avoir à apprendre de nouveaux concepts. Ceci n'est possible qu'à travers l'adoption d'un même modèle de représentation interne durant la totalité du processus de conception dont la sémantique évolue au fur et à mesure du raffinement (ce qui correspond au concept de *"continuité du modèle"*). Un tel modèle doit être capable de fournir les concepts de tous les niveaux d'abstraction et de capturer de manière formelle les résultats intermédiaires des différentes étapes du processus de conception. La maintenance de la continuité du modèle est une caractéristique importante puisqu'elle facilite d'une part la conception sans erreur (pas de déformation ou perte d'information liée à une transcription de modèle) et d'autre part améliore la traçabilité lors de la vérification et la validation du système conçu.

Une autre caractéristique essentielle du modèle de représentation interne est qu'il doit être interprété indifféremment comme logiciel ou matériel ("modèle unifié"). Une telle représentation unifiée des parties logicielles et matérielles au sein d'un même modèle permettra d'envisager le partitionnement de manière efficace ainsi qu'un développement conjoint qui élimine les problèmes d'intégration des différentes parties du système conçu. Le besoin d'unification du processus de développement est donc nécessaire non seulement pour spécifier sous le même modèle des sous-systèmes aux implémentations différentes (matériel ou logiciel), mais aussi pour unifier les diverses activités nécessaires à travers les différentes phases de développement sous la même représentation interne (ce qui correspond au concept de *"conception unifiée"*).

D'autres caractéristiques importantes du modèle nécessitent d'être mentionné, notons en particulier :

- Son origine : chaque modèle de représentation repose sur un modèle d'exécution qui spécifie le flot de données et/ou le flot de contrôle. Selon la classification faite par Varea [58], le modèle peut appartenir soit à la classe des modèles développés initialement pour les systèmes dominés par le contrôle et qui ont été étendus pour inclure le flot de données (noté M_{CD}) ou à la classe des modèles développés initialement pour les systèmes orientés flot de données et qui ont été étendus pour inclure le flot de contrôle (noté M_{DC}) ou en fin à la classe des modèles développés spécialement pour supporter une combinaison aussi bien du flot de contrôle que du flot de données (noté $M_{\bar{b}}$). Partant de l'évidence que le meilleur modèle est celui qui traduit le mieux les traitements de l'application du système, la connaissance de l'origine du modèle est donc un facteur très utile pour mieux guider le choix du modèle le plus approprié (concept *"natural fit"*).
- Sa capacité à offrir des propriétés telles ques : la puissance de modélisation (concurrence "parallélisme", communication, données et temps...), les possibilités d'analyse (possibilité de simulation, vérifiabilité, disponibilité d'outils...), la possibilité de synthèse (implantation en matériel

et en logiciel, disponibilité d'outils et méthodes supportant le modèle, capacités d'automatisation...), le support de la complexité, l'abstraction et la hiérarchie...

- **Exploration efficace de l'espace de conception :** tout au long du processus de conception et en particulier lors de l'étape de partitionnement matériel/logiciel, on cherchera le compromis matériel/logiciel le mieux adapté aux critères de performances et de coût visés (i.e la solution architecturale qui satisfait d'une façon optimale les contraintes imposées).
  Ces compromis (solutions architecturales) ne peuvent être examinés qu'à travers une exploration des différentes alternatives de conception. Cependant, étant donné la complexité des systèmes mixtes matériel/logiciel, et l'éventail des possibilités offertes par la technologie, le nombre des alternatives de conception à examiner est extrêmement grand. Face à la taille exponentielle de l'espace des solutions et pour ne pas être fastidieux, le choix du meilleur compromis matériel/logiciel doit être le plus automatisé possible (partitionnement automatique) avec une exploration efficace de l'espace des solutions architecturales. Cette exploration, qualifiée d'efficace, doit permettre la recherche de solution optimale (meilleur compromis matériel/logiciel) dans un délai raisonnable à l'échelle humain sans parcourir pour autant l'ensemble de l'espace des solutions. Ceci n'est possible qu'à travers la restriction, au moyen de contraintes, de l'espace des solutions à explorer à l'ensemble des solutions qui sont correctes vis a vis des contraintes imposées, en particulier en coupant les fausses pistes menant à des minima locaux sans intérêt [92]. L'objectif étant de réduire considérablement le nombre de solutions à examiner tout en augmentant la probabilité de trouver la solution optimale.
- **Réutilisation aisée :** Il est généralement admis que pour appréhender la complexité grandissante des systèmes matériel/logiciel, il faut augmenter les possibilités de réutilisation aussi bien des applications que des architectures. Pour ce faire, la tendance des nouveaux environnements de conception conjointe consiste à adopter un modèle de construction basé sur une séparation nette entre les deux vues : fonctionnelle (algorithmes de l'application), architecturale (architecture matérielle d'implantation) dès les premières étapes du cycle conception-implantation. Ce modèle de construction particulier (appelé *"Y-chart approach"*) repose sur un environnent permettant de différencier la spécification des algorithmes décrivant le comportement de l'application, de la spécification de l'architecture supportant leurs implantation et de la spécification de l'étape d'implantation proprement dite de l'application sur une architecture particulière. Cette séparation des modèles autorise aussi bien le portage des algorithmes et architectures que leur réutilisation : une même application peut être réutilisée sur une nouvelle architecture, ou être transformée et réimplantée sur une architecture existante. Ceci favorise une exploration efficace de l'espace des implantations possibles de l'algorithme sur l'architecture et par là même une réduction considérable du délai de la recherche de la meilleure implantation (l'adéquation algorithme-architecture) ainsi qu'une arrivée plus rapide du produit final sur le marché.
- **L'utilisation de lois et de méthodes formelles :** La complexité croissante des systèmes conçus par des approches de conception conjointe rend de plus en plus souhaitable, voire indispensable, l'utilisation de lois et de méthodes formelles tout au long du processus de conception et pas uniquement pour la vérification/validation à posteriori. L'objectif étant d'offrir des cadres formels qui permettent d'effectuer des techniques d'analyse statique de correction et de preuve formelle de propriétés ainsi que des simulations mixtes (logiciel/matériel) à différents niveaux d'abstraction

du cycle de conception-implantation assurant ainsi une validation continue tout au long de la conception.

Il est clair que l'on ne construit pas un système (un avion par exemple) pour vérifier ensuite qu'il fonctionne (vole) bien ; on le construit essentiellement en ayant en permanence à l'esprit, depuis le début de sa conception jusqu'à sa réalisation finale, qu'il doit fonctionner (voler) suivant certains paramètres ou lois et cette propriété est constamment "prouvée" et totalement "intégrée" au processus de réalisation à tous les niveaux.

C'est pourquoi, la seule méthode raisonnable pour atteindre ces objectifs est de recourir à des approches de conception conjointe basées sur des méthodes formelles permettant d'effectuer des analyses statiques de correction, indispensables pour assurer un développement rigoureux et systématique depuis la spécification de haut niveau jusqu'à la génération automatique ou la réalisation des programmes/circuits qui les implémentent.

– **L'interfaçage aux outils de synthèse matérielle et de compilation :**
Pour assurer une exploitation automatique, et profiter de la technologie existante et très performante dans le domaine de la compilation et de la synthèse de circuits, il faut offrir des systèmes de conception conjointe capable de s'interfacer facilement aux principaux outils de CAO et compilateurs pour des machines mono ou multi-processeurs. Ce choix est imposé pour diverses raisons et considérations (maturité indiscutable de ces outils créés pour résoudre des problèmes bien précis, leur maintenance et amélioration constante, etc.).

2.5 Bilan sur les systèmes de conception conjointe

L'étude de cette liste d'environnements, qui ne se veut pas être exhaustive mais plutôt représentative du domaine, a permis de relever les constatations suivantes :

– Au niveau spécification : on constate que les environnements de co-design suivent l'une des deux approches soit une spécification homogène ou une spécification hétérogène.
Les environnements basés sur une "spécification hétérogène" tel que CoWare utilisent des langages spécifiques pour les parties matérielles et logicielles, chaque partie du système est ainsi spécifiée à l'aide d'un langage correspondant à sa nature. Cette spécification différente pour le matériel et le logiciel peut présumer à l'avance des parties qui doivent être implantées en matériel et de celles qui doivent être en logiciel. De plus, cela pose des problèmes et des difficultés qui surviennent lors des étapes ultérieures de la conception, où le système doit être unifié sous une représentation interne pour pouvoir effectuer les opérations de partitionnement, de simulation etc...
Les environnements basés sur une "spécification homogène" utilisent un langage unique pour la spécification aussi bien des parties matérielles que des parties logicielles du système global. Plusieurs environnements (Cosyma, Vulcan, Polis...) tentent de suivre cette approche. Néanmoins, afin de permettre à la spécification de fournir une combinaison plus cohérente des différents concepts nécessaires à la modélisation des systèmes mixtes matériel/logiciel, la spécification est souvent faite à un niveau d'abstraction moins élevé. L'élévation du niveau d'abstraction des spécifications entraîne une augmentation du fossé entre les concepts utilisés pour les spécifications et ceux utilisés pour la synthèse du matériel.

Pour se faire certains environnements ont cherché à utiliser des versions étendues des langages tels que C ou Java pour modéliser les concepts matériels et les concepts de la communication comme Cosyma qui accepte une spécification en langage Cx, une extension du langage C, l'environnement Vulcan qui adopte une autre extension du langage C appelée HardwareC. On recense cependant peu d'environnements qui ont réussi à offrir une spécification homogène capable de modéliser un système à un niveau plus élevé. Les exemples de tels environnements sont : Polis qui utilise ESTEREL, et COSMOS qui utilise SDL. Cependant, le modèle sous-jacent à ces deux langages est basé sur le modèle de machines d'états finis (FSM *Finite-State-Machine* qui souffre du problème d'explosion d'états dès que la taille du système à spécifier est relativement grande.

– Du point de vue de l'approche adoptée pour le partitionnement, on constate que dans la majorité des environnements, les techniques de partitionnement sont relativement simples et varient selon le type de spécification en entrée : nous distinguons ceux qui commencent avec une spécification entièrement logicielle ("approche homogène") et effectuent une migration de certaines parties critiques du code vers le matériel jusqu'au respect des contraintes de performances comme Cosyma, de ceux qui commencent avec une spécification entièrement matérielle ("approche homogène") et effectuent une migration des parties non critiques vers le logiciel, réduisant ainsi le coût de l'application comme Vulcan. D'autres ne se limitent pas à un type de spécification en entrée ("approche hétérogène") et les différentes fonctionnalités d'une spécification sont affectées à des implantations logicielles ou matérielles en fonction des contraintes de façon automatique comme dans (*COSYMA*, *CoWare*, *GRAPE II*, ...) ou de façon manuelle comme dans (*Ptolemy*, *Polis*, ...). Signalons qu'aucune méthode ne s'impose comme meilleure que les autres à ce jour. Chacune est en effet bien adaptée au problème particulier qu'elle cherche à résoudre. Ceci est dû, en partie, à la jeunesse des travaux de recherche traitant le problème du partitionnement ainsi qu'à sa complexité intrinsèque.

– Du point de vue de l'architecture cible choisie pour supporter et exécuter le système mixte, nous distinguons ceux qui ciblent des architectures multiprocesseurs composées de plusieurs processeurs logiciels et matériels comme Ptolemy, Coware,... de ceux qui ciblent des architectures monoprocesseurs composées généralement d'un micro-processeur exécutant le logiciel et d'un ensemble de co-processeurs matériels (ASICs (ou FPGAs)) jouant le rôle d'accélérateurs pour la partie logicielle comme Polis, Vulcan, Cosyma,.... Il est clair que la satisfaction des contraintes technologiques et économiques d'une part ainsi que l'évolution constante de la technologie favorisent les environnements de conception conjointe supportant des architectures mixtes multiprocesseurs.

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

Enfin, du point de vue global les approches les plus anciennes (Vulcan et Cosyma) ne disposent pas de cycle de conception complet, et leurs niveaux d'abstraction est trop bas (HardwareC et C^x

respectivement). Ils sont tous les deux basés sur des approches orientées plateforme : Vulcan présente une extension (migration) vers le logiciel et Cosyma une extension (migration) vers le matériel. D'autres approches présentent des cycles complets de conception tels que Polis et Ptolemy pour des types de systèmes spécifiques mais souffrent encore de quelques lacunes. En effet, bien que Ptolemy apparaît être l'outil le plus mûr, en particulier par sa capacité à intégrer le plus grand nombre de modèles de calcul, ce qui offre aux concepteurs différents niveaux d'abstraction pour la spécification fonctionnelle, il est cependant limité à la cosimulation et sa capacité de co-synthèse matérielle/logicielle reste limitée. Il ne supporte pas une séparation nette entre les deux vues fonctionnelle et architecturale, ce qui réduit les possibilités de réutilisation et ne fournit qu'une implantation du système spécifié basée sur la machine virtuelle Java. L'environnement Polis quant à lui présente une méthodologie formelle unifiée supportant la spécification séparée selon le modèle en "'Y"' et il est spécialement efficace pour les systèmes à contrôle intensif. Cependant, son modèle de spécification basé sur les CSFMs restreint son utilisation à ce type particulier de systèmes et ne permet pas son utilisation pour les systèmes à forte dominante de données. L'explosion combinatoire des états relative au modèle CFSMs limite aussi son utilisation aux systèmes de taille réduite. De plus, l'architecture qu'il cible est une architecture mono-processeur particulière.

De manière générale, chacun des environnements de conception conjointe est dédié à un domaine d'applications particulier auquel il est particulièrement adapté, et aucun n'est capable de supporter la conception d'applications de domaines différents comme par exemple les applications de traitement de signal et les applications à contrôle intensif. Ceci est dû principalement au manque d'un modèle commun pour une représentation correcte et efficace aussi bien des flots de données que des flots de contrôle. De ce fait, chaque environnement est basé sur le modèle le plus approprié à son domaine d'application et offre ainsi une approche de conception-implantation performante et particulièrement adaptée à la résolution des problèmes spécifiques posés par le domaine d'applications auquel il est dédié. Cette diversité d'objectifs rend très difficile toute tâche d'évaluation comparative indépendante du domaine d'application.

Nous tentons tout de même de récapituler dans le tableau suivant 2.1 les caractéristiques jugées les plus importantes des différents environnements de conception conjointe présentés dans la section précédente. Ces caractéristiques portent principalement sur l'approche de co-synthèse adoptée (ex. le support ou pas du modèle en "Y", ligne 3), l'origine du modèle interne s'il y en a (flot de données et/ou flot de contrôles ligne 1), le support du partitionnement automatique (ligne 4) et la possibilité offerte par l'approche à explorer efficacement l'espace de conception (ligne 5), le niveau d'abstraction de la méthodologie qui est quantifié en fonction du niveau d'abstraction du langage de spécification utilisé (ligne 6). Nous qualifions les différents niveaux de moyen, faible ou élevé et restons ainsi en harmonie avec Jerraya [93] qui utilise aussi ces termes. Nous mettons le point sur les capacités de l'approche à assurer la génération de code autant pour la synthèse du logiciel, du matériel que pour la synthèse des interfaces (ligne 8) et sur le fait que l'approche soit fondée sur un modèle formel aussi bien au niveau des spécifications qu'au niveau des règles de transformations appliquées tout au long du processus de conception (ligne 7).

À travers cette étude des approches existantes, nous pouvons constater que bien que la plupart des environnements intègrent plus ou moins les fonctionnalités et les caractéristiques souhaitables d'un système de conception conjointe, aucun d'entre eux offre la gamme complète de ces fonctionnalités.

Critère d'évaluation	**Ptolemy**	**Cosyma**	**Vulcan**	**Polis**	**Cosmos**	**CoWare**	**GRAPEII**
Spécification	*Graphe flot de données*	*C**	*HardwareC*	*Esterel, FSM*	*SDL*	*DFL, C,..*	*DFL*
Modèle interne : origine	-	*ES-graphe CDFG : M_{CD}*	*CDFG : M_{DC}*	*CFSMs : M_{CD}*	*Solar : M_{CD}*	-	*M_D*
Support du modèle "Y-Chart"	*Non*	*Non*	*Non*	*Oui*	*Non*	*Oui*	*Oui*
Support du partitionnement automatique ?	*Non*	*Non, semi-automatique*	*Oui, par migration*	*Non, manuel*	*Non, semi-automatique*	*Non*	*Oui*
Support de l'exploration de l'espace de conception ?	*Oui*	*Non*	*Non*	*Oui(Y-chart like)*	-	*Non*	*Oui,(Y-chart like)*
Niveau d'abstraction de l'approche	*variable*	*faible*	*faible*	*très élevé*	*élevé*	*élevé*	*moyen*
Approche formelle	*Non*	*Non*	*Non*	*Oui*	*Non*	*Non*	*Non*
Support pour la synthèse	*Non*	*Non-*	*Non-*	*Oui*	*Oui*	*Non*	*Oui*
Architecture ciblée	*Architecture paramétrable (Mono ou Multi-processeurs*	*Mono-processeur : CPU + coprocesseur*	*Mono-processeur : CPU + ASIC avec un bus*	*Mono-processeur*	*Multi-processeurs*	*Multi-processeurs avec des ASICs*	*Multi-processeurs avec des FPGAs*

TAB. 2.1 – Environnements de conception conjointe

En fait, aucun environnement n'offre un flot de conception *automatisé* et *unifié* basé sur une approche *formelle* favorisant l'exploration et la réutilisation à travers la *séparation nette* entre la vue fonctionnelle et la vue architecturale tout en offrant un flot de conception *sans rupture* basé sur une continuité du modèle. Ce dernier inconvénient est principalement dû à la non-maintenance de la cohérence du modèle durant la totalité du processus.

2.6 Conclusion

Ce chapitre a présenté les principaux concepts et étapes de la conception conjointe. Cette introduction nous a permis d'appréhender le contexte actuel de la conception conjointe afin de mieux pouvoir situer la méthodologie AAA présentée dans le chapitre suivant.

L'état de l'art présenté nous a permis de bien cerner les problématiques liées au domaine de la conception conjointe, mais il a aussi mis en évidence l'inadéquation entre les approches et les environnements existants et les exigences attendues. En effet, malgré le nombre de plus en plus grand d'environnements et de travaux dans le co-design, l'automatisation de l'ensemble du processus de co-design, partant de la spécification à l'implantation finale du système est loin d'être un problème résolu.

Comme aucun des environnements n'est capable de satisfaire simultanément à tous les critères de qualité des outils de conception conjointe (approche formelle automatique, conception unifiée, continuité du modèle,....), nous nous sommes convaincus de la nécessité d'essayer de combler ce manque. En effet, notre contribution tente de combler les manques décrits ci-dessus en travaillant sur une extension de la méthodologie AAA développée à l'INRIA-Rocquencourt. Le grand avantage de cette méthodologie est l'utilisation d'un même modèle pendant toutes les étapes de la conception ainsi que l'automatisation du flot de conception depuis la spécification jusqu'à l'implantation. Cela offre un avantage que la majorité des approches de codesign connues n'offrent pas : la possibilité de maintenir la cohérence du modèle, et des informations qu'il encapsule, tout au long du processus de conception. Ceci est grandement facilité par le choix du modèle de graphe comme modèle de cospécification.

Comme nous verrons par la suite, l'intégration de cette extension de la méthodologie AAA au logiciel *SynDEx* qui la supporte nous permettra d'avoir un outil de conception conjointe très performant par rapport aux environnements existants.

Chapitre 3

La méthodologie AAA d'adéquation algorithme-architecture

La méthodologie AAA d'Adéquation Algorithme-Architecture développée par l'INRIA dans le projet OSTRE est le cadre principal dans lequel se place notre travail. Nous y consacrons ce chapitre afin de présenter son flot de conception-implantation, ses différents modèles de spécification et de transformation, ainsi que son processus de génération du code embarqué. Cette présentation méthodologique se termine avec une brève présentation du logiciel de CAO SynDEx qui supporte la méthodologie AAA. Le but étant de permettre de comprendre les tenants et aboutissants de la méthodologie et de ses modèles et ensuite de mettre en avant l'intérêt du travail présenté dans ce manuscrit et son apport pour la méthodologie.

3.1 Introduction

L'état de l'art présenté dans le chapitre précédent a mis en évidence le besoin d'aborder la conception sous l'angle de l'exploration, l'estimation et le prototypage, en développant des méthodologies de conception qui privilégient l'adéquation entre les algorithmes des applications visées et les architectures matérielles supportant leurs implantations. Cette adéquation consiste à concevoir l'architecture cible conjointement à l'algorithme décrivant le comportement de l'application, en favorisant l'adaptation mutuelle de l'algorithme et de l'architecture en vue de la réalisation d'une implantation optimisée. Cependant, la problématique de l'adéquation algorithme-architecture est en général un problème d'optimisation complexe visant à mettre en correspondance de manière efficace l'algorithme et l'architecture pour réaliser une implantation optimisée. Ceci conduit globalement à étudier conjointement les aspects algorithmiques et architecturaux afin de déterminer quelle est l'architecture d'un système complet la mieux adaptée à un traitement et, à l'inverse, comment reformuler l'algorithme de l'application, voire le dégrader, pour faciliter son implantation optimisée.

Plus précisément, le problème se pose d'abord de trouver l'architecture la mieux adaptée à un algorithme donné satisfaisant à certaines contraintes. On choisit ainsi une solution logicielle programmée en utilisant un ou plusieurs processeurs ou/et une solution matérielle câblée sous la forme d'un ASIC ou/et de un ou plusieurs FPGA. Cependant, lorsque l'on ne peut pas toujours de cette manière satisfaire les contraintes, on est parfois conduit à modifier l'algorithme lui-même. On doit soit modifier sa granularité, ce qui amène simplement à un redécoupage de l'algorithme, soit modifier plus radicalement sa structure (reformulation complète). Ce processus itératif dans lequel l'algorithme influence l'architecture et réciproquement est l'essence même du concept de l'adéquation algorithme-architecture.

Il est clair qu'un tel processus consistant à spécifier l'algorithme en fonction des propriétés architecturales et à concevoir l'architecture en fonction des propriétés algorithmiques autorise aussi bien le portage des algorithmes et des architectures que leur réutilisation : une même application peut être réutilisée sur une nouvelle architecture, ou être modifiée et ré-implantée sur une même architecture. Ce qui favorise une exploration efficace de l'espace des implantations possibles et, par là même, une réduction considérable du délai de l'obtention de la meilleure implantation (adéquation algorithme-architecture) ainsi qu'une arrivée plus rapide du produit final sur le marché. De telles approches communément qualifiées d'adéquation algorithme-architecture conduisent ainsi à développer des méthodologies, plus ou moins formelles, permettant de poser des problèmes d'optimisation pour dimensionner au mieux les architectures de circuits intégrés spécifiques et/ou de multi-processeurs et, enfin, de générer automatiquement du code efficace tout en réduisant les temps et les coûts de développement de l'application étudiée et en évitant toute rupture entre les différentes phases du cycle de développement.

Dans cette optique, l'équipe OASTRE de l'INRIA-Rocquencourt (Institut National de Recherche en Informatique et en Automatique) a développé une méthodologie formelle d'adéquation algorithme-architecture, nommée AAA, particulièrement adaptée pour les systèmes distribués temps réel embarqués [38]. Cette méthodologie de prototypage rapide et d'implantation optimisée, est le cadre principal où se place notre travail, c'est pourquoi nous rappelons dans la suite de ce chapitre ses principes ainsi que sa démarche.

Avant de présenter les principes généraux de cette méthodologie AAA, nous avons jugé nécessaire de préciser tout d'abord le sens que l'on donnera dans le cadre de la méthodologie AAA aux notions d'application, d'algorithme, d'architecture, d'adéquation.

- L'application dans le cadre d'AAA désigne un système composé de deux sous-systèmes en constante interaction. Le premier sous-système correspond à l'environnement physique de l'application qu'il faut contrôler. Le second correspond à un système réactif qui doit réagir aux variations d'états du premier, lui même composé d'un ensemble d'algorithmes spécifiant l'application et d'une architecture matérielle supportant l'exécution de ces algorithmes issus principalement du domaine du contrôle-commande et du traitement du signal et des images.
- L'algorithme (ou plus précisément l'algorithme réalisant l'application) est le résultat de la transformation de la spécification fonctionnelle du système réactif sous sa forme plus ou moins formelle, en une spécification logicielle adaptée à un traitement numérique. Il désigne l'organisation de transformations et traitements à opérer sur les données, dénommé communément "programme ou logiciel". Ce terme générique d'algorithme est utilisé plutôt que celui de programme ou logiciel afin d'assurer une indépendance vis à vis des langages de programmation.
- L'architecture est la partie matérielle du système réactif. Elle désigne le matériel qui décrit la structure d'exécution physique permettant de réaliser les traitements. Par abus de langage le terme architecture est souvent utilisé dans le cadre d'AAA dans son sens générique pour signifier à la fois le calculateur réalisant les traitements lui-même ainsi que ses caractéristiques structurelles.
- L'implantation assure la mise en oeuvre (*"spatial and temporal mapping"*) de l'algorithme sur l'architecture. C'est-à-dire l'allocation des ressources matérielles de l'architecture aux opérations de l'algorithme, l'ordonnancement des opérations allouées à une même ressource, puis la compilation, le chargement, et enfin le lancement de l'exécution du programme correspondant sur le calculateur, avec dans le cas des processeurs le support d'un système d'exploitation (souvent appelé *exécutif* quand il offre des services temps réel) dont le surcoût ne doit pas être négligé.
- L'adéquation consiste à mettre en correspondance de manière efficace l'algorithme et l'architecture pour réaliser une implantation optimisée. Par abus de langage, cette notion d'implantation optimisée est utilisée bien qu'on ne puisse pas garantir l'obtention d'une solution optimale pour le type de problème, de complexité NP-difficile, auquel est appliquée AAA. Nous nous contenterons donc d'une solution approchée obtenue rapidement, plutôt que d'une solution exacte obtenue dans un temps rédhibitoire à l'échelle humain à cause de la complexité combinatoire exponentielle de la recherche de la solution exacte (optimale).

3.2 Méthodologie AAA

La méthodologie "Adéquation Algorithme Architecture" (AAA) proposée par l'INRIA-OSTRE vise le prototypage rapide et l'implantation optimisée d'applications distribuées temps réel embarquées devant être tolérantes aux fautes, telles celles rencontrées en contrôle-commande de systèmes complexes comprenant du traitement du signal et des images. Cette méthodologie est fondée sur une approche globale formalisant l'algorithme et l'architecture à l'aide de graphes et l'implantation de l'algorithme sur l'architecture à l'aide de transformation de ces deux graphes. Le graphe flot de données modélisant l'algorithme au niveau comportemental est transformé progressivement jusqu'à ce qu'il corresponde au graphe matériel modélisant l'architecture. Ces transformations représentent une distribution (allocation spatiale) et un ordonnancement (allocation temporelle) des calculs sur les processeurs et des communications sur les liaisons physiques inter-processeurs. Le résultat de ces transformations est un

exécutif temps réel distribué supportant l'implantation de l'algorithme spécifié sur l'architecture donnée. L'adéquation revient ainsi à choisir parmi toutes les implantations possibles d'un algorithme sur une architecture, l'implantation dont les performances, déduites des caractéristiques des composants de l'architecture, respectent les contraintes temps réel, tout en minimisant la consommation de ressources, ce qui se traduit par un problème d'optimisation.

L'intérêt principal du modèle de graphe utilisé réside dans sa capacité à exprimer tout le parallélisme, non seulement dans le cas de l'algorithme (parallélisme potentiel que renferme l'algorithme, exprimé à travers un graphe d'algorithme : graphe flot de données) et de l'architecture (parallélisme disponible qu'offre effectivement l'architecture, exprimé à travers un graphe matériel : graphe de ressources de calcul, de ressources mémoire et de ressources de communication), mais aussi dans le cas de l'implantation de l'algorithme sur l'architecture (réduction du parallélisme potentiel au parallélisme disponible exprimée par transformations de graphes, c'est-à-dire distribution et ordonnancement des calculs et des communications).

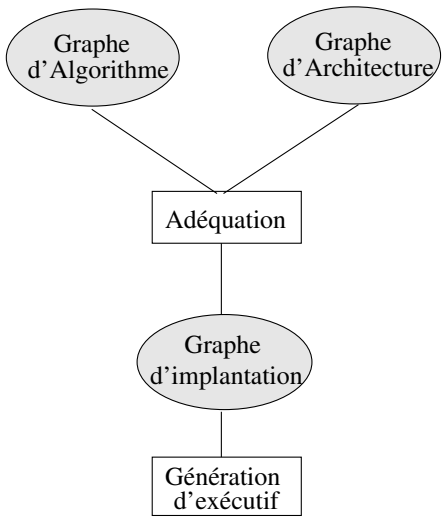


FIG. 3.1 – Flot d'implantation *AAA*

Notons que cette modélisation unifiée de l'algorithme, de l'architecture et de l'implantation à base de graphe offre un cadre formel bien adapté à la conception sans erreur et à la traçabilité (pas de déformation ou perte d'information liées à une transcription du modèle). Une telle unification et continuité du modèle permet de vérifier facilement que toutes les transformations conservent les propriétés d'ordre partiel du graphe d'algorithme initial, que l'implantation finale correspond bien à la spécification initiale et que les optimisations effectuées seront conservées lors de l'exécution réelle. Ceci évite d'une part toute rupture entre la phase de spécification et celle de réalisation et d'autre part la rupture entre le prototypage rapide de l'application et la réalisation industrielle de série.

On notera également que la démarche AAA [38][42], schématisée sur la figure 3.1, est conçue sur un modèle de construction en "Y" (*Y-chart model*) réalisant la spécification séparée de l'algorithme décrivant le comportement de l'application, de la spécification de l'architecture supportant son implantation et de la spécification de l'implantation proprement dite de cet algorithme sur une architecture

particulière. Ce qui autorise la portabilité et la réutilisabilité aussi bien des algorithmes que des architectures et favorise ainsi l'exploration efficace de l'espace des solutions en vue de la recherche de la meilleure mise en correspondance de l'algorithme sur l'architecture.

Par la suite, nous allons aborder comment à partir d'un algorithme, et d'une architecture on arrive à construire rapidement une implantation optimisée grâce à une approche globale d'Adéquation Algorithme-Architecture. Nous présentons successivement le modèle d'algorithme, le modèle d'architecture, le modèle d'implantation et enfin le logiciel de CAO niveau système SynDEx concrétisant AAA.

3.3 Modèle d'algorithme

Un algorithme est le résultat de la transformation de la spécification fonctionnelle d'une application en une spécification adaptée à son traitement numérique. Plus précisément un algorithme, tel que défini par Turing [94], est une séquence (ordre total) finie d'opérations directement exécutables par une machine à états finie (calculateur). Dans la littérature, il existe deux approches principales de modélisation d'algorithme à base de graphes orientés qui diffèrent selon la sémantique des arcs et de l'exécution des sommets du graphe : l'approche graphe flot de données et l'approche graphe flot de contrôle.

- Dans un graphe flot de données [43], chaque sommet représente une opération (fonction de calcul) et chaque arc représente une dépendance de donnée entre deux opérations. Les arcs entrant dans un sommet représentent des données d'entrée d'une opération et les arcs sortant représentent les données de sortie résultant du traitement effectué. L'exécution d'un sommet se produit quand toutes ses données d'entrée sont disponibles, ainsi les opérations sont exécutées dans un ordre résultant seulement des dépendances de données. Donc, au niveau de la spécification d'un algorithme, la notion de variable n'existe pas. Plus généralement, chaque hyperarc (diffusion d'une donnée) représente une précédence d'exécution causée par un transfert (dépendance) de données entre un sommet producteur et un ou plusieurs sommets consommateurs (figure 3.2). Deux sommets opérations qui n'ont pas à se transférer des données ne sont pas connectées par un arc. L'ensemble des arcs définit donc un ordre partiel d'exécution sur les opérations (établi par les dépendances des données entre opérations) qui traduit naturellement du "parallélisme potentiel". Un arc peut représenter aussi une dépendance de contrôle, la donnée qu'il véhicule est alors une donnée de contrôle utilisée pour gérer par exemple une itération ou du conditionnement. L'ordre sur les opérations à exécuter et l'ordre dans lequel ces opérations lisent ou écrivent les données sont cohérents.

  Cette spécification d'ordre est très importante dans le cas des algorithmes de contrôle-commande, de traitement de signal et des images où il est fondamental de maîtriser l'ordre dans lequel on manipule les données [39].
- Dans un graphe flot de contrôles, chaque sommet représente une opération qui consomme et produit des données dans des variables pendant son exécution, et chaque arc représente une précédence d'exécution. L'ensemble des arcs représente l'écoulement du contrôle entre les opérations et définit donc un ordre total d'exécution des opérations plus restrictif que celui du flot de données. Il n'y a pas de relation entre l'ordre des opérations à exécuter et l'ordre dans lequel ces opérations lisent ou écrivent les données dans les variables. Le flot de contrôle met seulement l'accent sur

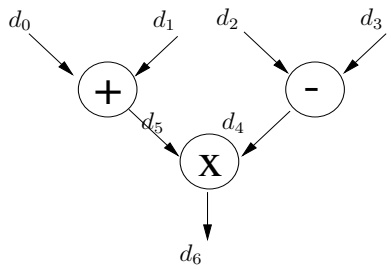


FIG. 3.2 – Exemple simple d'un graphe flot de données

l'ordonnancement des opérations en exigeant qu'une opération finisse avant que d'autres débutent mais il ne vérifie pas implicitement la disponibilité des entrées lorsqu'une opération commence. Les entrées sont déterminées pendant l'exécution des opérations, mais ne sont pas exigées pour commencer une opération.

Un organigramme est un exemple classique de graphe flot de contrôle habituellement utilisé avant d'écrire un programme dans un langage impératif (tels que Fortran, COBOL, Basic, C, Ada, Java, ...).

Notons que le flot de contrôle définit le traitement (ou calcul) en termes d'ordre total d'exécution des opérations, ce qui correspond bien à la définition standard donnée plus haut d'un algorithme. De ce fait, si on veut faire apparaître du parallélisme potentiel dans l'approche flot de contrôle, pour pouvoir exécuter des opérations en parallèle, il faut faire une analyse de dépendance des données communiquées entre les opérations par l'intermédiaire des variables communes, afin de pouvoir décomposer le graphe initial en plusieurs graphes flot de contrôle composés en parallèle qui se partageront les variables communes via des transferts de données nécessaires. Dans le cas général où l'on a des opérations calculées de façons alternatives selon une valeur de conditionnement le problème est NP-difficile.

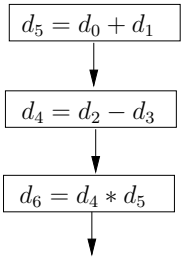


FIG. 3.3 – Exemple simple d'un graphe flot de contrôle

Un exemple simple de graphe flot de contrôle, correspondant au graphe flot de données de la figure 3.2 est montré sur la figure 3.3. Dans ce graphe on voit que l'on a imposé à l'opération "+" de s'exécuter avant l'opération "-" alors que celles-ci pourraient s'exécuter potentiellement en parallèle, comme cela est fait dans le graphe flot de données de la figure 3.2.

Pour modéliser l'algorithme, et pour mettre en évidence son parallélisme potentiel, la méthodologie AAA utilise le modèle de graphe flot de données. L'adoption de ce modèle plutôt que le graphe flot de contrôle a été motivée pour les raisons suivantes :

- La représentation en graphe flot de données permet la description explicite des dépendances de données, et par conséquent la mise en évidence du parallélisme potentiel entre les opérations nécessaires à l'implantation parallèle de l'algorithme de l'application, alors que, dans le cas d'un graphe flot de contrôle, il faut extraire ces dépendances, implicitement décrites par le partage de variables entre opérations.
- Dans un graphe flot de contrôle, les données sont indépendantes du contrôle, c'est à l'utilisateur qui effectue la spécification d'assurer la cohérence entre l'ordre de consommation et de production des données à travers l'accès aux variables, et l'ordre d'exécution des opérations qui manipulent ces données. L'établissement d'un ordre entre les données peut s'avérer difficile et peut conduire à des erreurs lorsque par exemple on cherche à réutiliser les variables. En revanche, dans le cas du flot de données, les données sont dépendantes du contrôle et l'ordre d'accès aux données est ainsi directement imposé par l'ordre d'exécution des opérations. Ce type de spécification d'algorithme est moins sujet à des erreurs, ce sera au compilateur des langages qui supporte ce modèle flot de données d'assurer l'ordre d'écriture et de lecture dans les variables, qui pourront ainsi être réutilisées sans erreurs.

En résumé, le modèle d'algorithme d'AAA est un graphe basé sur le modèle flot de données supportant la notion d'ordre partiel d'exécution qui d'une part permet d'exprimer le parallélisme potentiel pour pouvoir l'exploiter (à l'aide de transformations simples) lors de l'implantation distribuée, et d'autre part prend en compte l'interaction infiniment répétitive des applications cibles avec leurs environnements (systèmes réactifs) ainsi que l'aspect conditionnel des algorithmes. L'algorithme est donc spécifié par un hypergraphe orienté acyclique infiniment répété et conditionné [32], appelé Graphe Factorisé et Conditionné de Dépendances de Données (GFCDD), dont les sommets sont des opérations partiellement ordonnées (parallélisme potentiel) par leurs dépendances de données. Ce modèle étendu introduisant le flot de contrôle à l'intérieur du flot de données en prenant en compte les spécificités des parties répétitives (boucles), ainsi que le cas de calculs exécutés conditionnellement (if..then..else), sera étudié plus en détail dans le chapitre suivant.

3.4 Modèle d'architecture

Dans le cadre de la méthodologie AAA, l'architecture matérielle distribuée supportant l'exécution temps réel de l'algorithme de l'application est dite *multicomposant* car sa structure offrant du parallélisme effectif comprend en général des composants capteurs et actionneurs, des composants programmables (processeurs RISC, CISC, DSP, microcontrôleurs) et des composants non programmables (circuits intégrés spécifiques ASIC, circuits FPGA reconfigurables), interconnectés par des médias de communication. Cette multiplicité des composants peut être nécessaire pour satisfaire un besoin de puissance de calcul, de modularité ou de localisation des capteurs et des actionneurs par rapport aux traitements à réaliser.

Les modèles les plus classiquement utilisés, dans la littérature, pour spécifier de telles architectures parallèles ou distribuées peuvent être classés en deux grandes familles : les modèles de haut niveau (PRAM, DRAM, BSP, LogP, CGM, etc) et les modèles de bas niveau (basés sur les Hardware Description Languages, niveau RTL). L'étude de ces modèles, établi dans le cadre de la thèse de Thierry

Grandpierre [33], a mis en évidence leurs principales limites dans le cadre d'AAA : soit ils sont trop abstraits dans le sens où des hypothèses trop simplificatrices sont effectuées sur l'architecture sous-jacente (durée de communication constante...), ce qui ne permet pas de refléter précisément le comportement de l'architecture et par conséquent d'effectuer une prédiction de performances fiable en vue de l'optimisation, soit leur granularité est trop fine comparée à celle de la spécification algorithmique ce qui a aussi pour conséquence d'augmenter à la fois la complexité de spécification de l'architecture et la complexité du problème d'optimisation de l'implantation.

Pour cela AAA utilise un modèle d'architecture multicomposant hétérogène [33, 34] basé sur les graphes orientés d'un niveau de granularité intermédiaire à ceux cités ci-dessus, qui est suffisamment précis pour permettre la prédiction de performance et la génération automatique d'exécutif, mais suffisamment générique pour modéliser le plus grand nombre possible d'architectures avec différents types de communication (passage de message sur bus point à point ou multipoints, communication par mémoire partagée mono-port, multiports etc.). Ce modèle permet de mettre en évidence le parallélisme effectif offert par une architecture ainsi que toutes les ressources qu'il est nécessaire d'allouer pour exécuter un algorithme (processeurs, mémoires, canaux DMA).

L'architecture distribuée hétérogène est modélisée par un graphe orienté, dénoté $G_{ar} = (S, A)$, où S est l'ensemble des sommets de ce graphe et A l'ensemble de ses arcs. Chaque sommet de S est une machine à états finis (machine séquentielle) et chaque arc de A est une connexion orientée entre une sortie d'une machine séquentielle et une entrée d'une autre machine séquentielle, le graphe d'architecture forme ainsi un réseau d'automates [95]. L'ensemble des sommets S se décompose en quatre sous-ensembles correspondant chacun à un type de machines à états finis (FSM) : *l'opérateur* pour séquencer des opérations de calcul (séquenceur d'instructions) (S_{opr}), le communicateur pour séquencer des opérations de communication (canal DMA) (S_{com}), *le Bus/mux/démux avec ou sans arbitre* pour sélectionner, diffuser et éventuellement arbitrer des données (S_{bus}), *la mémoire* pour stocker des données et des programmes (S_M). Chaque arc c de A représente une connexion entre les entrées et les sorties des FSMs, le graphe d'architecture forme ainsi un réseau de composants (automate FSM) inter-connectés par des médias de communication (lien, bus, mémoires partagées ...). L'interconnexion de ces différents sommets ne peut pas être faite de n'importe quelle manière, il est nécessaire de respecter un ensemble de règles. Par exemple, deux opérateurs ne peuvent pas être connectés directement ensemble. Ils peuvent chacun être connecté à une RAM partagée ou à une SAM pour communiquer, en passant ou non par l'intermédiaire de communicateurs pour assurer le découplage entre calcul et communication.

On note également que l'hétérogénéité de l'architecture, dans le cadre d'AAA, ne signifie pas seulement que les sommets peuvent avoir chacun des caractéristiques différentes (par exemple : durée d'exécution des opérations et taille mémoire des données communiquées), mais aussi que certaines opérations ne peuvent être exécutées que par certains opérateurs. Pour cela, dans AAA, on associe à chaque opérateur et à chaque communicateur l'ensemble des opérations que chacun d'eux est capable de réaliser, et pour chaque opération sa durée, son occupation mémoire, la consommation associée à son exécution, etc. Cette caractérisation de l'architecture cible, nécessaire à la prédiction de performance d'une application, est d'une importance primordiale pour le processus de l'optimisation.

Ce modèle à base de graphe permet de faire des spécifications plus ou moins détaillées d'une même architecture en fonction de la précision avec laquelle on souhaite faire l'implantation optimisée. Il faut tout de même noter que plus l'architecture sera détaillée plus le problème d'optimisation sera long à

résoudre.

3.5 Modèle d'implantation

L'implantation d'un algorithme sur une architecture consiste à réaliser, en tenant compte des contraintes, une distribution et un ordonnancement des opérations de l'algorithme sur les opérateurs de l'architecture caractérisée.

La distribution, appelé aussi placement ou répartition, est la première étape du processus d'implantation. Elle consiste à affecter chaque opération de l'algorithme à un opérateur capable de l'exécuter. Ceci conduit à une partition du graphe de l'algorithme initial G_{al}, en autant d'éléments de partition (sous-graphes) qu'il y a d'opérateurs dans le graphe de l'architecture G_{ar}. Il faut ensuite affecter chaque élément de cette partition, c'est-à-dire chaque sous-graphe du graphe de l'algorithme initial à un opérateur du graphe de l'architecture. Puis il faut effectuer une répartition des dépendances de données du graphe de l'algorithme reliant des opérations appartenant à des éléments de partition différents en autant d'éléments de partition qu'il y a de routes ("chemins") dans le graphe de l'architecture. Il faut ensuite affecter chaque élément de partition de dépendances de données à une route. Pour chacune de ces dépendances de données inter-partitions, il faut créer autant de nouvelles opérations de communication qu'il y a de médias de communication sur la route sur laquelle elle est affectée, et affecter ensuite ces opérations de communication aux communicateurs correspondants de la route.

L'ordonnancement, deuxième étape de l'implantation, consiste à rendre total (linéariser) l'ordre partiel formé par le sous-graphe des opérations affectées à un opérateur, car celui-ci est une machine séquentielle dont le rôle est d'exécuter séquentiellement des opérations. Il consiste aussi à rendre total l'ordre partiel formé par le sous-graphe des opérations de communications affectées à un communicateur, car celui-ci est aussi une machine séquentielle dont le rôle est d'exécuter séquentiellement des opérations de communication. Pour cela on ajoute, si nécessaire, des précédences d'exécution entre les opérations d'un même sous-graphe en vérifiant que l'ordre total ainsi créé inclut l'ordre partiel du graphe de l'algorithme initial.

Une implantation est donc le graphe résultat d'une transformation du graphe de l'algorithme (ajout des opérations de communication et des dépendances d'ordonnancement), influencée par le graphe de l'architecture. Elle peut être mathématiquement formalisée, en intention, comme la composition de trois relations binaires : le routage, la distribution et l'ordonnancement ; chacune d'elles mettant en correspondance deux couples de graphes (algorithme, architecture) [32]. Ou encore comme une loi de composition externe où un graphe d'algorithme est composé avec un graphe d'architecture pour donner comme résultat un graphe d'algorithme transformé (distribué et ordonnancé).

Pour un couple donné de graphes d'algorithme et d'architecture, on comprend aisément qu'il existe un nombre fini, mais qui peut être grand, de distributions et d'ordonnancements possibles. En effet, on peut effectuer plusieurs partitions du graphe de l'algorithme, en fonction du nombre d'opérateurs du graphe de l'architecture, et pour chaque sous-graphe affecté à un sommet opérateur il y a plusieurs linéarisations possibles de ce sous-graphe.

Parmi toutes ces implantations possibles, ayant chacune des performances différentes, une implantation particulière appelée *implantation optimisée* sera choisie en fonction des contraintes temps réel et des ressources matérielles utilisées, ce qui se traduit par un problème d'optimisation. On obtient ainsi

un graphe d'implantation optimisé.

3.6 Optimisation de l'implantation : Adéquation

La recherche d'une implantation optimisée d'un algorithme sur une architecture respectant les contraintes temps réel et minimisant la taille de l'architecture, correspond à une adéquation ("mise en correspondance efficace" entre cet algorithme et cette architecture). Notons que par abus de langage, cette notion d'implantation optimisée est utilisée bien que l'on ne peut pas garantir l'obtention d'une solution optimale pour ce type de problème reconnu NP-difficile : en effet, pour un couple donné de graphes d'algorithme et d'architecture, il existe un nombre très vaste mais fini de graphes d'implantations possibles. Le nombre de cas à étudier étant très élevé, l'adéquation se contentera donc d'une solution approchée obtenue rapidement, plutôt que d'une solution exacte obtenue dans un temps rédhibitoire à l'échelle humaine à cause de la complexité combinatoire exponentielle de la recherche exhaustive de la solution exacte (optimale).

La méthodologie AAA utilise alors des heuristiques qui sont à la fois rapides et donnant des résultats proches de la solution optimale. Pour répondre aux besoins du prototypage rapide des applications temps réel de contrôle-commande comprenant du traitement du signal et des images, il est intéressant de choisir des heuristiques rapides, afin de pouvoir essayer de nombreuses variantes d'implantation en fonction du coût et de la disponibilité des composants et de tester rapidement l'impact de l'ajout de nouvelles fonctionnalités.

C'est pourquoi AAA privilégie les heuristiques gloutonnes (c'est-à-dire sans remise en cause des solutions partielles intermédiaires conduisant à une solution finale ; elles sont aussi dites sans retour arrière) et plus particulièrement celles de liste car ce sont elles qui donnent le plus rapidement leur résultat tout en ayant une bonne précision [96]. Celle qui a été formalisée et automatisée [32] est basée sur un algorithme de liste glouton, associé à une fonction de coût qui prend en compte la durée d'exécution des opérations et des communications pour indiquer le degré d'urgence qu'il y a à distribuer et ordonnancer une opération sur un opérateur. La description détaillée de cet algorithme n'est pas l'objet de ce chapitre. Nous indiquons simplement que cet algorithme basé principalement sur les dates d'exécutions des calculs, requiert une phase préalable de caractérisation dans laquelle une durée d'exécution est associée à chaque opération de calcul ou resp. de communication. Cette durée est fonction de l'opérateur ou resp. du communicateur qui est capable de l'exécuter.

3.7 Génération d'exécutifs distribués temps réel

Dès qu'une distribution et un ordonnancement ont été déterminés par l'heuristique, il est assez simple de générer automatiquement des exécutifs distribués temps réel, principalement statiques avec une partie dynamique uniquement quand cela est inévitable (calcul des booléens de conditionnement à l'exécution) [35]. Il faut encore insister ici sur le fait que dans les applications embarquées de traitement du signal et des images, l'exécutif qui supportera l'exécution distribuée temps réel doit être le moins coûteux possible. C'est pourquoi AAA évite autant que faire se peut les exécutifs dynamiques, qui s'ils semblent faciles à mettre en oeuvre pour l'utilisateur, conduisent à une surcouche logicielle système, laquelle prend à l'exécution des décisions d'allocation de ressources (changements de contexte pour

allouer le temps CPU à différentes tâches, gestion de files d'attente pour les communications, codage-transfert-décodage d'informations de routage, etc.), dont le surcoût n'est pas négligeable.

Avec l'approche préconisée, le processus de génération d'exécutif statique, taillé sur mesure pour l'application, est systématique. Il se fait selon des règles décrivant la transformation d'un graphe d'implantation optimisé en un graphe d'exécution [33][35]. Pour chaque opérateur (resp. chaque communicateur) on construit un programme séquentiel formé de la séquence des opérations de calcul (resp. des opérations de communication) qu'il doit exécuter. Les opérations de communications sont des "Send" et des "Receive" de données transmises entre communicateurs via une SAM (communication par passage de message), ou des "WRITE" et des "READ" quand les données sont transmises via des RAM (communication par mémoire partagée). Pour garantir les précédences d'exécution entre les opérations appartenant à des séquences de calcul et/ou de communication différentes, et pour garantir l'accès en exclusion mutuelle aux données partagées par les opérations de ces séquences, on ajoute des opérations de synchronisation avant et après chaque opération qui lit (resp. écrit) une donnée écrite (resp. lue) par une opération appartenant à une autre séquence. Ces opérations de synchronisation utilisent des sémaphores générés automatiquement. Notons que ce mécanisme simple de sémaphores permet de conserver les propriétés d'ordre de l'algorithme, éventuellement montrées avec les langages synchrones, car elles ont été conservées lors du choix de la distribution et de l'ordonnancement optimisés.

Il y a autant d'exécutifs générés, chacun d'eux correspondant à un fichier distinct, qu'il y a d'opérateurs dans l'architecture. Chaque fichier d'exécutif est un code intermédiaire indépendant de l'opérateur, c'est-à-dire du processeur puisqu'il n'y a qu'un opérateur par processeur, composé d'une liste d'appels de macros qui seront traduites par un macro-processeur en autant de programmes dans le langage source préféré (C, ou assembleur par exemple). Chacun de ces programmes sources sera compilé puis chargé dans la mémoire programme du processeur correspondant. Les définitions de macros qui sont dépendantes de l'opérateur (du processeur) peuvent être classées en deux ensembles. Le premier ensemble est un jeu extensible de macros applicatives réalisant les opérations de l'algorithme. Le second ensemble, appelé noyau d'exécutif, est un jeu fixe de macros système qui supportent le chargement initial des mémoires programmes, la gestion mémoire (allocation statique, ...), le séquencement (sauts conditionnels et itérations...), les transferts de données inter-opérateurs (macro-opérations de communication transférant le contenu de macro-registres), les synchronisations inter-séquences (assurant l'alternance entre écriture et lectures de chaque macro-registre partagé entre séquence de calcul et séquences de communication), et le chronométrage (pour permettre la mesure des caractéristiques des opérations de l'algorithme et des performances de l'implantation).

En résumé, le processus de génération d'exécutifs se résume en deux étapes principales. La première traduit la distribution et l'ordonnancement produits par l'heuristique d'optimisation (i.e graphe d'exécution) en un macro-code intermédiaire générique, c'est-à-dire indépendant du langage cible préféré pour chaque type de processeur. La seconde traduit le macro-code intermédiaire en code source compilable par le compilateur spécifique à chaque type de processeur cible. Cette seconde étape est basée sur le macro-processeur m4 (standard sous Unix, version GNU), complété pour chaque type de processeur cible par un fichier de définitions de macros spécifiques à ce type de processeur. Ce choix de scinder le processus de génération d'exécutifs en deux étapes, l'une indépendante de l'architecture et l'autre dépendante, a été principalement motivé dans la perspective d'avoir un générateur qui s'adapte facilement à différents types de processeurs cibles.

3.8 Logiciel de CAO niveau système *SynDEx*

SynDEx est un logiciel de CAO niveau système qui concrétise la méthodologie AAA pour le prototypage rapide et l'optimisation de l'implantation d'applications distribuées temps réel embarquées.

C'est un logiciel graphique interactif qui prend en entrée une spécification de l'algorithme de l'application sous la forme d'un graphe factorisé et conditionné de dépendances de données ainsi qu'une spécification de l'architecture-cible sous la forme d'un graphe d'opérateurs (processeurs) et de média de communications. *SynDEx* exécute ensuite des heuristiques d'optimisation, réalisant l'adéquation entre l'algorithme et l'architecture. *SynDEx* effectue cette distribution et cet ordonnancement optimisé (adéquation) de façon statique. Il assure que la spécification de l'algorithme et la spécification de l'architecture conduiront à une implantation correcte sur une machine multi-processeur. Après visualisation de la prédiction des performances temps réel de cette distribution/ordonnancement, un exécutif temps réel est généré automatiquement, permettant l'exécution de l'algorithme sur l'architecture et libérant l'utilisateur des tâches lourdes de programmation bas niveau. Le générateur d'exécutifs de *SynDEx* transforme le graphe flot de données qui spécifie l'application en un graphe flot de contrôle codé par l'exécutif. Cet exécutif sans interblocage est construit, avec un surcoût minimal, à partir d'un noyau d'exécutif dépendant du processeur cible. Actuellement *SynDEx* supporte les architectures multiprocesseur à base de stations de travail UNIX, de processeurs i80x86, de processeurs de traitement du signal TMS320C40, TMS320C60 et ADSP21060, de microcontrôleurs MPC555, MC68332 et i80C196. Des noyaux pour d'autres processeurs sont facilement portés à partir des noyaux existants.

Le logiciel de CAO *SynDEx* est utilisé aussi bien par des universitaires que par des industriels. Il est distribué gratuitement sur le web à l'url : http ://www.syndex.org.

3.9 Extension d'AAA aux circuits intégrés spécifiques

Certaines applications de traitement du signal et plus particulièrement de traitement d'images demandent une puissance de calcul croissante quand elles doivent s'exécuter sous contraintes temps réel. Cette puissance ne peut être fournie que par le recours à des architectures multicomposants. Ces architectures sont réalisées avec des composants programmables (RISC, DSP), qui offrent une grande flexibilité et/ou composants non programmables (ASIC, FPGA, CPLD), qui offrent de bonnes performances. De telles architectures mixtes permettent donc d'effectuer de manière performante certaines parties du traitement et de ce fait d'améliorer la performance globale du système en vue de satisfaire les contraintes imposées.

La méthodologie AAA développée à l'INRIA considère que dans les architectures multicomposants réalisées à partir de composants programmables (processeur) et non programmables (ASIC, FPGA), les composants non programmables ne sont capables d'exécuter qu'une unique opération alors qu'un composant programmable peut en exécuter plusieurs, en cela il est plus flexible bien qu'en général il soit moins rapide en termes d'exécution qu'un composant non programmable. Cependant à l'état actuel AAA ne couvre pas les composants non programmables (ASIC, FPGA) et fait l'hypothèse que ces composants ont été conçu par ailleurs.

Le but de nos travaux de recherche est de permettre de concevoir aussi les composants non programmables (circuits intégrés spécifiques) avec *SynDEx* dans le cadre de la méthodologie AAA. Ce travail d'extension de la méthodologie AAA que nous proposons consiste à définir une méthodologie de

conception appropriée à ces composants circuits et à établir les bases de son intégration dans le logiciel *SynDEx*. L'intégration de cette extension d'AAA aux circuits intégrés dans l'environnement *SynDEx* le placera au rang des outils de conception conjointe matériel-logiciel de systèmes embarqués hétérogènes temps réel, puisqu'il sera capable de générer des implantations optimisées pour les architectures à base de processeurs et/ou de circuits intégrés spécifiques, ces derniers ayant eux-même été conçus et réalisés avec l'extension de *SynDEx* aux circuits (*SynDEx-IC*).

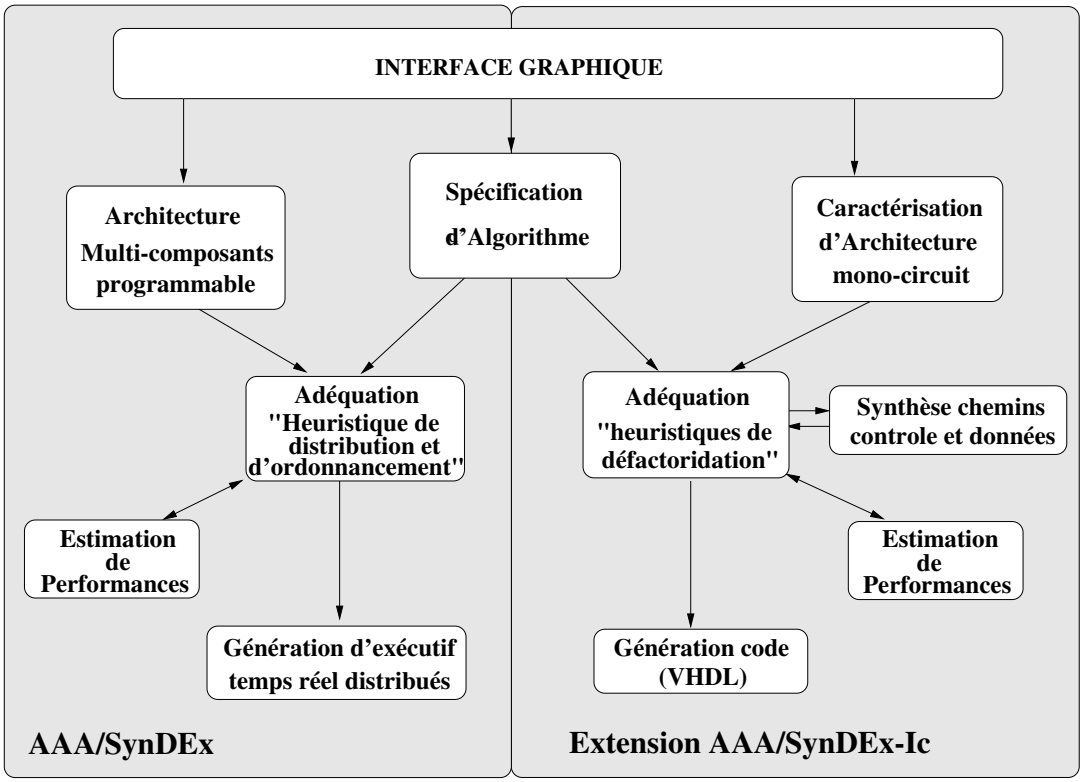


FIG. 3.4 – Extension d'AAA/SynDEx

Notons que dans l'AAA (composants programmables), il faut réaliser l'adéquation entre un graphe d'algorithme et un graphe d'architecture pour obtenir un graphe d'implantation puis d'exécution optimisés.

Dans l'extension AAA circuits que nous proposons, la technique d'implantation consistera donc en une transformation du graphe d'algorithme en chemins de données et de contrôle du mono-circuit correspondant. Contrairement à l'implantation sur composants programmables, comme nous ne nous intéressons dans un premier temps, qu'aux implantations matérielles mono-circuit le graphe de l'algorithme n'est pas ici influencé par un graphe d'architecture, en effet celui-ci n'est pas spécifié. Ainsi le graphe d'algorithme est transformé directement en un graphe d'implantation qui est le graphe d'architecture du circuit visé.

Dans le cadre de la conception conjointe logiciel/matériel, le choix de la partie de l'algorithme qui sera implantée en logiciel c'est-à-dire sur composants programmables, et celle qui sera implantée en matériel

c'est-à-dire sur un ou plusieurs circuits intégrés spécifiques est généralement fait de façon manuelle. Comme nous avons un cadre formel unique pour décrire l'algorithme, l'architecture processeurs et/ou circuit il est plus facile de poser un problème d'optimisation pour le partitionnement logiciel-matériel, afin que ce dernier puisse s'effectuer de manière automatique et non plus manuelle. On peut ainsi espérer des résultats nouveaux dans ce domaine.

3.10 Conclusion

La méthodologie AAA d'adéquation algorithme-architecture que nous avons présentée offre un cadre formel et un flot de conception unifié sans rupture permettant d'une part de faire des vérifications temporelles en termes d'ordre sur les événements qui entrent et sortent de l'algorithme, éliminant ainsi un grand nombre d'erreurs concernant la logique de l'algorithme, et d'autre part d'aider à la recherche, parmi toutes les implantations possibles que l'on peut faire d'un algorithme celle qui respecte les contraintes temps réel et minimise les ressources matérielles. Enfin elle permet de générer automatiquement des exécutifs taillés sur mesure pour chacun des processeurs de l'architecture, qui sont sans interblocage et dont le surcoût est très faible, en assurant que les vérifications faites précédemment restent valides. Tout ceci place AAA ainsi que le logiciel SynDEx qui la supporte parmi les méthodes et outils logiciels d'aide à l'implantation les plus avancés dans le domaine de développement d'applications distribuées temps réel embarquées.

Étant donné le besoin en puissance de calcul de ces applications temps réel, et la variété des algorithmes à implanter, nous sommes contraint à utiliser lors de l'implantation en complément des processeurs, des circuits intégrés spécialisés (ASIC ou FPGA). Toutefois, le flot d'implantation supporté par cette méthodologie concerne les architectures cibles de types multicomposant dans lesquelles processeurs et circuits intégrés spécifiques ont été conçus par ailleurs. La contribution de ce travail consiste à l'étendre à la conception de circuits intégrés spécifiques.

Les travaux d'extension présentés dans cette thèse permettront de concevoir les circuits intégrés spécialisés dans le cadre de la méthodologie AAA, ce qui permettra d'avoir un cadre formel unique pour poser le problème de la conception conjointe logiciel-matériel (co-design). Le chapitre suivant sera ainsi consacré à une étape primordiale du flot d'extension qui est la spécification algorithmique : point de départ du processus d'implantation matérielle sur circuits.

Chapitre 4

Modèle d'algorithme

Ce chapitre traite la première partie du flot d'extension d'AAA qui est la spécification algorithmique. Après un état de l'art des différents modèles utilisés dans la littérature nous justifions le choix du modèle de flot de données comme modèle de base de spécification dans AAA. Nous faisons le point sur l'intégration des structures de contrôle dans ce modèle en particulier l'aspect conditionnel et nous présentons nos enrichissements.

4.1 Introduction

La conception de système est souvent perçue comme un procédé allant d'une spécification fonctionnelle à son implantation finale à travers une séquence de tâches de raffinements ou de transformations. Ces tâches intermédiaires sont généralement caractérisées par une transformation d'une spécification plus abstraite en une spécification plus détaillée afin d'aboutir enfin à une spécification détaillée qui peut être implantée. Au début de ce processus, seules les fonctionnalités du système à concevoir ainsi que les contraintes qu'il doit respecter sont connues. La première étape consiste alors à spécifier ces fonctionnalités tout en capturant toutes les informations nécessaires à sa bonne exécution (par exemple les contraintes architecturales et de fonctionnement), sans prendre de décision de mise en oeuvre ni introduire de biais vers l'une ou l'autre des implantations (logiciel ou matériel), afin d'assurer l'indépendance entre la description comportementale initiale et la réalisation matérielle finale.

Cette spécification initiale au plus haut niveau d'abstraction est déterminante et conditionne pour une grande part tout le reste du processus. En effet la flexibilité, la modularité et l'extensibilité du flot de conception-implantation sont influencées par les propriétés du formalisme (modèle ou langage) de spécification utilisé. C'est pourquoi l'utilisation de formalismes, dont la sémantique est définie de manière précise, à partir de modèles mathématiques bien adaptés, s'avère être incontournable.

L'approche de spécification indépendante des langages en utilisant un modèle de représentation (appelé *approche "modèle"*) permet de faire reposer la démarche de conception sur des bases solides et s'avère être de plus en plus une solution attractive. L'avantage le plus important de l'utilisation d'un modèle de spécification des systèmes est de pouvoir d'une part faire évoluer sa sémantique au fur et à mesure du raffinement et d'autre part d'offrir un cadre formel parfaitement établi à la conception sans erreur permettant d'effectuer les vérifications formelles le plus tôt possible dans le cycle de développement de l'application en garantissant que l'implantation finale correspond bien à la spécification initiale selon le principe "correct par construction". Il permet aussi de développer des outils automatiques d'aide à l'implantation sans se soucier des évolutions des langages de spécification.

Le modèle étant le formalisme de représentation interne sur lequel doivent être appliquées toutes les étapes du processus de conception, le déroulement du processus de conception est considérablement influencé par le modèle de spécification adopté. Le choix de ce modèle est donc d'une importance primordiale et il dépend fortement du type de système à implanter.

Comme nous nous intéressons dans le cadre d'AAA aux systèmes temps réel comprenant des algorithmes de contrôle commande et de traitement du signal et des images, le modèle de spécification adopté est un modèle de graphe factorisé et conditionné des dépendances de données particulièrement adapté aux spécificités de ces applications en particulier : le traitement régulier et irrégulier de données en traitement du signal et des images, l'interaction infiniment répétitive de l'application avec son environnement,...

L'étude et la présentation de ce modèle feront l'objet de ce chapitre, que nous commencerons par une courte présentation des modèles utilisés dans quelques uns des principaux travaux sur la conception conjointe des systèmes mixtes.

4.2 Principaux modèles de spécification

Dès 1997, De Micheli [46] soulevait le problème de manque de modèles bien définis pour modéliser les systèmes matériel/logiciel et en faisait une des causes retardant les progrès dans le domaine de la conception conjointe. De nombreux travaux ont été menés depuis dans le domaine de la modélisation de systèmes. La compréhension de ces modèles, et leur rapport aux différents niveaux d'abstraction ainsi qu'aux langages de spécification est bénéfique pour les concepteurs de ces systèmes afin de faciliter le choix du modèle pour la spécification en vue de la création de modèles exécutables et vérifiables. Ce choix dépend fortement du type de système à concevoir. Il est souvent guidé par plusieurs considérations en particulier les caractéristiques du système à concevoir : le type de traitements dans l'application, sa possibilité de décrire des spécifications non-fonctionnelles (contraintes,..), l'abstraction, la concurrence (parallélisme), la hiérarchie, la disponibilité et l'efficacité des outils d'analyse et de synthèse, la preuve formelle de propriétés, synthèse/compilation...

4.2.1 Les automates à états finis (FSM : Finite State Machine)

Les automates à états finis classiques constituent le modèle le plus utilisé pour les systèmes comportant une forte dominante de contrôle [7]. Les FSM consistent en un ensemble d'états (ou modes), un ensemble d'entrées, un ensemble de sorties, une fonction qui définit les sorties en termes d'entrées et d'états, et une fonction *prochain-état* déterminant l'état suivant [7]. Visuellement, les FSM sont représentés comme un ensemble de noeuds et d'arcs. Les noeuds représentent les états du système et les arcs les transitions ou plus précisément le transfert de contrôle entre ces états. Du fait de leur nature finie, les FSM se prêtent bien à l'analyse formelle et par conséquent, aux activités de co-vérification. L'un des inconvénients des FSM simples est qu'ils sont plats, ne permettant ainsi aucune décomposition hiérarchique des systèmes à modéliser. Cela entraîne donc une augmentation du nombre d'états et de transitions, ce qui rend ce modèle inutilisable dès que la complexité des systèmes à modéliser devient importante. Par ailleurs, les FSM simples ne permettent pas de modéliser les comportements concurrents (parallélisme). En outre, une petite variation dans la spécification peut produire un grand changement de l'automate. Par conséquent, les FSM simples peuvent s'avérer très peu pratiques pour modéliser des systèmes complexes.

Des extensions et des variantes du modèle typique les FSM simples ont été proposé pour remédier à ces lacunes, citons en particulier : les CFSM (Co-Design Finite State Machines), proposé par Chiodo et al [9] et utilisés dans l'environnement Polis [77], qui constituent une extension hiérarchique des automates à états finis classiques dédiés principalement aux systèmes orientés contrôle présentant une complexité algorithmique relativement faible. Les StateCharts introduits par Harel [10] constituent aussi une extension des FSMs en autorisant la hiérarchie et la concurrence et en permettant l'expression de gardes sur les transitions. Bien qu'ils permettent de modéliser de manière assez simple les automates complexes, étant donné qu'ils sont basés sur l'hypothèse synchrone, ce formalisme ne fait pas l'unanimité et provoque quelques divergences au sein de la communauté scientifique sur un certain nombre de constructions syntaxiques, sur des situations pouvant donner lieu à un indéterminisme, ou sur la sémantique du formalisme. Les SpecCharts [11] sont une autre extension des FSM classiques qui offrent comme StateCharts des mécanismes pour modéliser la hiérarchie, la concurrence. Outre les extensions apportées aux FSM et dont nous avons cité quelques unes des plus connues dans ce qui précède, il est

possible d'utiliser les automates à états finis en combinaison avec d'autres modèles de calcul afin de surmonter leurs faiblesses.

4.2.2 Les réseaux de Petri (RDP)

Les réseaux de Petri qui ont été conçus par C.A. Petri en 1962 [12] sont largement référencés pour la modélisation dans différents domaines scientifiques. Classiquement, un réseaux de petri est composé de quatre éléments de base : un ensemble de places, un ensemble de transitions, une fonction d'entrée qui associe les transitions aux places "application d'incidence avant" et une fonction de sortie qui est aussi une application des transitions aux places "application d'incidence après". La théorie mathématique, développée à travers plusieurs années, définissant leurs structures et leurs règles de transition en ont fait un outil de modélisation efficace [13, 15]. Ainsi, il est possible de déterminer à priori les caractéristiques et les propriétés du système modélisé, en particulier déterminer si le réseau est vivant (sans blocage). Notons également deux propriétés intrinsèques intéressantes des réseaux de Petri qui sont leur possibilité d'exprimer de la concurrence et leur nature asynchrone. La première caractéristique (le parallélisme) signifie que les événements peuvent se produire d'une manière indépendante s'ils sont rendus actifs. La propriété d'asynchronisme signifie qu'il n'y a pas de mécanisme d'horloge inhérent pour activer les transitions. Certaines communautés prétendent, cependant, que l'inconvénient des réseaux de Petri de base provient du manque de décomposition hiérarchique, du manque de mécanismes pour exprimer les contraintes temporelles ou du manque d'interprétation conduisant à une expression déficiente des calculs. Plusieurs extensions ont été apportées aux RDP classiques. Citons en particuliers : Les ETPN (Extended Timed Petri Nets : Réseaux de Petri Temporisés Étendus) qui sont des RDP étendus avec des informations temporelles pour faciliter l'évaluation des performances [16]. Les réseaux de Petri hiérarchiques (HPNs [17] : Hierarchical Petri Nets) dont la principale motivation est la difficulté de spécifier et de comprendre les graphes de réseaux de Petri de systèmes complexes par des représentations planes. Les HPN héritent des propriétés les plus importantes des réseaux de Petri (y compris la concurrence et l'asynchronisme).

4.2.3 La modélisation par processus communicants

Dans le modèle de processus séquentiels communicants (CSP :*Communicating Sequential Process*) [19], le système est modélisé comme un ensemble de processus séquentiels, qui fonctionnent de manière concurrente et qui communiquent les uns avec les autres à travers des canaux unidirectionnels en utilisant un protocole de synchronisation. Ce modèle est particulièrement bien adapté pour la modélisation d'applications où le partage de ressources est un élément clé. Son inconvénient majeur est la difficulté de maintenir son déterminisme étant donné qu'il repose sur une sémantique de concurrence par entrelacement, qui par nature est non-déterministe. Les CSP de base n'incluent pas intrinsèquement la notion de temps, ce qui peut rendre difficile la collaboration avec des modèles qui en tiennent compte. Ils peuvent cependant être enrichis avec cette notion. Les travaux de Thomas et al [59][60] utilisent les CSP pour modéliser le comportement du système sous forme d'un ensemble de processus matériels ou logiciels, indépendants et communicants, décrits chacun de manière comportementale à l'aide d'un langage de description de haut niveau. Certains langages parallèles sont fortement basés sur ce modèle, en particulier Lotos et Occam.

4.2.4 Modèles à événements discrets

Ce modèle basé fondamentalement sur le temps se focalise sur l'occurrence des événements. Un système à événements discrets (DE : Discrete Event) [20] peut être défini comme un système à états discrets, contrôlé par les événements, c'est-à-dire que l'évolution de son état dépend entièrement de l'occurrence d'événements discrets asynchrones. Lee [24] donne une description mathématique formelle de tels systèmes. Un système consiste en un réseau de blocs connectés par des arcs orientés. Les blocs communiquent entre eux et avec l'environnement à travers des événements. Un événement est une action instantanée qui cause des transitions d'un état discret à un autre. Chaque événement a une valeur et il est associé à une étiquette temporelle qui indique l'instant de son occurrence dans le modèle [23]. L'ensemble de ces événements est trié selon les étiquettes temporelles et il est analysé dans un ordre chronologique : "exécution chronologique" ; le temps est donc une partie intégrante du modèle.

Ce modèle est très utilisé dans la modélisation des systèmes informatiques, des réseaux de télécommunications ou des réseaux de transport. Bien qu'il soit très utile pour la spécification des systèmes temps réel, son inconvénient majeur est son coût très élevé car il est nécessaire d'ordonnancer totalement tous les événements selon leurs temps d'occurrence, ainsi la modélisation de la concurrence reste difficile à supporter [21]. Notons que ce modèle à événements discrets peut être spécifié à partir des langages Verilog et VHDL et qu'un simulateur du modèle DE est intégré dans l'environnement Ptolemy.

4.2.5 Les modèles réactifs synchrones

Ce modèle basé sur la synchronisation de tous les événements par une horloge globale, est dédié à la modélisation des systèmes réactifs temps réel d'où son appellation. Un système décrit dans ce modèle consiste en des blocs communicants où le temps de réaction de chaque bloc et le temps de communication entre les blocs sont supposés être instantannés. Les systèmes ainsi modélisés réagissent instantanément en produisant leurs sorties de façon synchrone aux entrées. L'avantage de cette supposition (appelé l'hypothèse synchrone) est que de tels systèmes sont plus faciles à décrire et à analyser. En règle générale, la concurrence entraîne le non-déterminisme mais le modèle synchrone/réactif (SR) se distingue par le fait qu'il fait cohabiter concurrence et déterminisme.

Les modèles SR sont adaptés aux applications avec des logiques de contrôle concurrentes et complexes. Du fait que le synchronisme fort du modèle permet de travailler séparément sur les aspects de correction fonctionnelle et temporelle des systèmes, les applications temps réel à sécurité critique constituent de bon candidats [24]. Cependant, à cause de cette propriété du synchronisme fort, certaines applications sont sur-spécifiées avec ce modèle, ce qui limite les alternatives d'implantation et rend les systèmes distribués difficiles à modéliser. Un autre exemple intéressant d'application pour lequel le modèle réactif synchrone convient idéalement est celui de la gestion du protocole d'accès au média à jeton "token-ring" [25].

De nombreux langages synchrones basés sur ce modèle ont été développés, nous distinguons dans littérature deux grandes familles : la famille des langages synchrones flot de données tels que Signal [29] et Lustre [30] et celle des langages synchrones flot de contrôle tels que Esterel [31] et Argos [121].

4.2.6 Les modèles graphes flots de données

Ce modèle utilise des graphes orientés, où les sommets représentent les fonctions et les arcs les transferts de données entre les calculs réalisés par ces fonctions décrivant ainsi l'ordre d'exécution des calculs. Un sommet (fonction de calcul) est prêt à exécuter son calcul dès que toutes ses entrées seront disponibles (arguments d'entrée), il consomme ainsi les données reçues sur chacun de ses arcs en entrée et les combine pour produire des données sur chacun de ses arcs de sortie, ainsi les fonctions sont exécutées dans un ordre résultant seulement des dépendances de données. Donc, chaque arc représente une précédence d'exécution induite par une dépendance (transfert) de données entre un sommet producteur et un ou plusieurs sommets consommateurs. L'ensemble des arcs définit un ordre partiel d'exécution traduisant naturellement du parallélisme potentiel. Ce modèle offre ainsi un schéma de programmes parallèles dans lequel le déterminisme est garanti. Les graphes flot de données sont fréquemment représentés de façon graphique et peuvent être hiérarchiques. Cela veut dire qu'un sommet du graphe flot de données (GFD) peut représenter un autre graphe orienté. Ces graphes flots de données sont très utilisés pour modéliser les systèmes à forte dominantes de données en particulier les applications de traitements de signal et d'images. Ptolemy, Cossap de Synopsys ou les travaux de Gupta sont des exemples d'utilisation de ce modèle dans le domaine de la conception conjointe. De nombreuses variantes du modèle graphes flots de données ont été utilisées pour représenter les systèmes mixtes matériel/logiciel, un aperçu général de ces différents modèles est présenté dans un papier de Lee [122]. Notons en particulier les réseaux de processus flots de données (*Dataflow process networks*) qui sont un modèle de calcul utilisé dans les systèmes de traitement de signaux. Les applications sont représentés par des graphes orientés où les noeuds (acteurs) représentent les calculs et les arcs (flots ou streams) représentent les séquences de données passés. Les sommets de calcul possèdent des règles d'activation permettant de déterminer leurs activations et l'opération spécifique à exécuter. Dans ce modèle, la communication des processus concurrents est faite par des canaux FIFO unidirectionnels non limités. Les modèles flots de données synchrones SDF (Synchronous Data Flow) conçus par Lee et al. constituent un cas particulier de ce modèle où le nombre d'échantillons de données consommés et produits par un noeud est connu statiquement (à la compilation).

En complément des modèles de spécification discutés ci-dessus, de nombreuses extensions et variantes de ces modèles existent dans la littérature. La liste des modèles que nous venons de présenter ne se veut donc pas être exhaustive mais plutôt représentative des différentes classes des modèles de spécification existants.

4.2.7 Choix d'un modèle de spécification

Comme cela a été souligné lors de l'introduction, le choix judicieux du modèle adopté pour la spécification d'un système est d'une importance primordiale, car il peut avoir une influence directe sur toutes les autres étapes de conception. Ce choix est souvent guidé par l'adéquation des propriétés et caractéristiques du modèle avec les spécificités du système à concevoir. Dès lors, l'évaluation des différents modèles doit donc être établie en fonction des exigences des systèmes mixtes matériel/logiciel en termes de déterminisme et de sûreté de fonctionnement, de distribution et de parallélisme, ainsi que de la rapidité de la réaction (complexité et synchronisme). Ces notions, qui constituent quelques-unes des préoccupations parmi les plus importantes des concepteurs d'applications temps réel, sont précisément

Modèles	*Application principale*	*Hiérarchie*	*Concurrence*	*Formelle*	*Non-déterminisme*	*Synchrone*
Automates à états finis (FSM)	*Orienté contrôle*	*Non* [a]	*Non*	*Oui*	*Oui*	*Non*
Réseaux de Petri (PN)	*Distribué à grande échelle*	*Non*[b]	*Oui*	*Oui*	*Oui*	*Non*
Processus communicants (CSP)	*Orienté données*[c]	*Oui*	*Oui*	*Non*	*Oui*	*Non*
Modèles à événements discrets (DE)	*Temps réel*	*Non*	*Non*	*Oui*	*Non*	*Non*
Modèles Synchrones/Réactives (SR)	*Réactive*	*Oui*	*Oui*	*Oui*	*Non*	*Oui*
Graphes flots de données (DFG)	*Traitement de signaux digitaux*	*Oui*	*Oui*	*Oui*	*Non*	*Non*

TAB. 4.1 – Modèles de spécification en conception conjointe

[a]Les extensions CFSM et Statecharts du FSM classique supportent la hiérarchie à un certain degré
[b]L'extension HPNs des PNs classiques supportent la hiérarchie à un certain degré
[c]En particulier, celle comprenant des problèmes de gestion et partage de ressources

des concepts fondamentaux qui forment le socle sémantique des modèles de spécification. Partant de ces constats, nous résumons et complétons également les diverses caractéristiques des modèles discutées ci-dessus en fonction de ces concepts dans le tableau récapitulatif 4.2.7. Les caractéristiques ainsi dressées correspondent à : l'application principale (quel type de systèmes est bien correctement représenté par le modèle), la hiérarchie (si la représentation supporte les structures hiérarchiques), la concurrence (si le modèle supporte la spécification des calculs concurrents), le fondement formel (si le modèle est basé sur une théorie mathématique qui défini clairement sa sémantique), le non-déterminisme (si le modèle de calcul offre une abstraction non-déterministe du comportement du système spécifié) et le synchronisme (si le modèle supporte une synchronisation de l'ensemble des éléments du système par une horloge globale).

Comme on peut le constater certains modèles sont plus appropriés à la modélisation des systèmes qui doivent manipuler de grandes quantités de données, d'autres modèles sont plus appropriés pour les systèmes comportant une forte dominante de contrôle. Ces modèles représentent donc plus efficacement soit les données, soit le contrôle mais prennent rarement en compte les deux aspects : les graphes flots de données (DFG), par exemple, sont bien adaptés à décrire les dépendances de données dans un système de traitement de signal, mais ne sont pas aussi si bien adaptés pour la modélisation de la logique de contrôle associée et la gestion des ressources. Les automates à états finis (FSM) sont très bien adaptés pour la modélisation d'une logique de contrôle plus au moins simple, mais sont moins bien adaptés à modéliser les dépendances de données et le calcul numérique. Les processus communicants (CSP) sont adaptés pour la gestion de ressource, mais ils sur-spécifient les dépendances de données. Ceci dit il est rare qu'un système réel ait un traitement exclusivement orienté données ou exclusivement orienté contrôle, ce qui explique la continuité des travaux de recherche visant à proposer de nouvelles solutions

pour une spécification complète des systèmes.

On constate aussi que parmi les principaux modèles de spécification cités, nombreux sont ceux qui ne prennent pas en compte la hiérarchie (FSM, PN,...). Cette insuffisance les rend inadéquats, dès que la taille et la complexité des systèmes à modéliser augmentent.

Quelques modèles ne permettent pas de prendre en compte la concurrence (FSM,DE,...) et introduisent de ce fait des contraintes qui empêchent le système spécifié de profiter pleinement des ressources disponibles dans l'architecture cible (les processeurs logiciels et les circuits matériels) qui doivent fonctionner en parallèle.

Signalons aussi que la majorité des modèles sont basés sur une sémantique mathématique forte, offrant ainsi des possibilités de vérification formelle et d'optimisations poussées, d'ailleurs c'est une propriété fortement exigée pour un modèle de spécification. On note également une prédominance de modèles asynchrones dans le sens où les transitions ne sont pas pilotées par un signal d'horloge.

En résumé, chacun des modèles présente des points forts pour la modélisation de certains concepts, mais aussi des points faibles pour la modélisation d'autres concepts, ce qui signifie qu'aucun modèle existant ne présente la totalité des capacités nécessaires à la modélisation de tous les types de systèmes à tous les niveaux d'abstraction. Ils offrent tous seulement des solutions partielles pour la spécification des systèmes actuels.

Actuellement, un modèle de spécification est essentiellement sélectionné en fonction des caractéristiques de l'application à implanter : pour un système qui répète périodiquement les mêmes transformations/opérations sur des paquets de données, comme un système de traitement du signal et des images, le modèle flot de données semble être le plus approprié. Par contre, pour un système qui n'effectue pas de calculs complexes, mais qui doit répondre à des séquences complexes d'événements externes, comme un système de contrôle-commande le modèle FSM est approprié. Pour les systèmes qui effectuent des transformations complexes de données, plusieurs fois en parallèle, comme les bases de données client-serveur ou les systèmes multi-tâches, le modèle CSP semble être le plus approprié [109].

Après avoir choisi le modèle approprié, il faut spécifier la fonctionnalité du système par l'intermédiaire d'un langage de spécification dont la sémantique est définie par le modèle choisi. VHDL et *Verilog* [2] sont des standards très utilisés, permettant de décrire les modèles CSP et FSM. *Esterel* est également utilisé pour décrire les modèles CSP et FSM. *StateCharts* permet la description de FSM hiérarchiques et concurrentes. *SpecCharts* décrit les modèles CSP, FSM hiérarchiques et concurrents. SDL [55] permet de décrire des GFD hiérarchiques contenant des FSM, *Silage* [66] décrit des GFD.

4.2.8 Motivation du choix d'un modèle flot de données

En conclusion de cette étude de l'existant nous pouvons affirmer que malgré les nombreux modèles existants aucun n'a réussi à faire l'unanimité, et encore moins à s'imposer comme standard. Jusqu'à présent, aucun consensus ne semble se dégager sur un modèle de spécification universel (idéal) qui soit utilisable pour toutes les classes d'applications. Pour le moment, la sélection d'un modèle approprié est souvent guidée par les caractéristiques de l'application à implanter. Néanmoins, il est généralement admis qu'un modèle qui se voudrait efficace pour modéliser les applications dans un processus de conception conjointe, doit présenter un certain nombre de caractéristiques. Il doit, en particulier :

– permettre de représenter aussi bien les flots de données que les flots de contrôle ;

- préserver la cohérence des différentes parties du système à travers toutes les étapes du processus (continuité du modèle) ;
- reposer sur un formalisme mathématique rigoureux, afin de permettre de recourir à la preuve formelle comme mécanisme de validation/vérification ;
- permettre de modéliser au maximum la concurrence (parallélisme) entre calculs ;
- prendre en compte la communication et la synchronisation ;
- permettre l'implantation facile en matériel ou en logiciel ;
- représenter la hiérarchie des systèmes complexes afin d'augmenter la lisibilité lorsque les systèmes modélisés sont de grande taille.

Partant de l'évidence que le meilleur modèle est celui qui traduit le mieux les caractéristiques du système qu'il modélise tout en satisfaisant au mieux des critères de qualité, le modèle de spécification adopté dans le cadre d'AAA est basé sur le modèle de flot de données. Ce choix est motivé par le fait que nous nous intéressons aux systèmes temps réel, dont les algorithmes de traitement du signal et des images des applications respectives sont caractérisés par des répétitions périodiques d'un ensemble d'opérations (sur des données différentes). Nous utilisons un modèle basé principalement sur les graphes flot de données (GFD) que nous appelons graphes factorisés et conditionnés de dépendances de données (GFCDD) afin de prendre en compte tous les aspects du contrôle y compris la répétition et l'alternative conditionnelle d'opérations.

Ce modèle de spécification constitue une extension de la version basique du modèle de graphe flot de données qui répond aux principales exigences de qualité d'un modèle de spécification. Basé au départ sur le modèle de flot de données son extension a permis d'introduire du flot de contrôle à l'intérieur du flot de données dans un même modèle de spécification. Le modèle flot de données permettant de décrire la concurrence, il s'adapte bien à décrire le parallélisme de l'application nécessaire à son implantation distribuée. Étant basé sur un modèle de graphe il permet naturellement de prendre en compte la hiérarchie : chaque opération du graphe peut être à son tour décrite par un sous-graphe permettant une spécification hiérarchique de l'algorithme jusqu'aux "opérations atomiques" que l'on ne peut spécifier à l'aide d'un sous-graphe. Cette "hiérarchie de graphes" offre des niveaux d'abstraction de la spécification permettant une souplesse dans la maîtrise de la complexité croissante des applications auxquelles nous nous intéressons. De plus, ce modèle repose sur une théorie mathématique bien fondée "la théorie des graphes et des ordres partiels", ce qui permet de démontrer formellement les propriétés de la spécification et de les maintenir durant tout le cycle de conception à travers des transformations de graphes.

4.3 Modèle de spécification AAA

Pour réaliser la spécification fonctionnelle d'un système, toutes les fonctions élémentaires décrivant la solution fonctionnelle sont à décrire complètement sous une forme algorithmique, de manière à exprimer avec exactitude leur comportement interne. Cette transformation de la spécification fonctionnelle sous sa forme plus ou moins formelle, en une spécification logicielle adaptée à un traitement numérique par calculateur conduit en général à un algorithme obtenu par composition de l'ensemble des algorithmes correspondant aux fonctions élémentaires de la solution fonctionnelle.

En principe, la spécification complète d'un tel algorithme exprimé sous forme d'opérations directement exécutables doit tenir compte d'une part de l'ordre d'exécution des opérations à effectuer, c'est-à-dire flot de contrôle, et d'autre part des données manipulées ou utiles ainsi que leur utilisation, c'est-à-dire flot de données.

Cependant, si la version basique du modèle graphe flot de données s'avère répondre aux principales exigences en spécification d'AAA en particulier l'expression et la mise en évidence du parallélisme potentiel de l'algorithme spécifié nécessaire à l'implantation parallèle distribuée, il ne permet pas de prendre en compte les structures de sélection et de répétition liées au flot de contrôle et qui sont nécessaires à la spécification complète de l'algorithme de l'application. En effet si le graphe flot de données traite la séquence avant l'ordre partiel défini par le graphe, il ne traite ni la séquence arrière ni le choix alternatif.

Pour remédier à cette lacune, et pour pouvoir offrir un modèle de spécification complet capable d'une part d'exprimer aussi bien le flot de données que tout le flot de contrôle dans un même modèle, et, d'autre part de prendre en compte l'interaction infiniment répétitive des applications cibles avec leurs environnements (systèmes réactifs) AAA utilise une variante de la version basique du modèle graphe flot de données appelé graphe factorisé et conditionné de dépendance de données. Ce modèle est décrit en détail dans [41, 28].

Tout d'abord, chaque système temps réel que l'on cherche à décrire dans le cadre de la méthodologie AAA est un système réactif, qui réalise une infinité de fois la séquence : Acquisition des entrées, Opération de calculs, Production des sorties (noté motif). Dès lors, pour faire apparaître au maximum le parallélisme potentiel tout en prenant en compte cet aspect réactif des applications visées, le modèle flot de données décrivant le motif est répété indéfiniment de manière spatiale modélisant ainsi l'interaction infiniment répétitive de l'application avec son environnement. Le repliement de ce graphe de dépendances de données acyclique infini suivant le motif (ou encore l'instance de l'interaction Entrées-Calculs-Sorties) conduit alors a une répétition temporelle du motif telle que l'infinité des capteurs et actionneurs (sommet sans prédécesseur et sommet sans successeur) est ramenée à un seul capteur et un seul actionneur réalisant temporellement leur acquisition et production de données (voir exemple présenté plus loin sur la figure 4.6).

Le sous-graphe résultant de cette factorisation du graphe infini par repliement spatial à son motif répété comprenant l'ensemble des opérations appartenant à la même itération (ou encore à la même séquence Entrées-Calculs-Sorties) et l'ensemble des dépendances de données entre ces opérations modélise donc *le graphe d'algorithme* de l'application spécifiée.

Comme il est rare qu'une application ait un comportement exclusivement orienté flot de données ou exclusivement orienté flot de contrôle, il est nécessaire que le modèle du graphe d'algorithme adopté pour la spécification soit capable d'exprimer aussi bien les aspects liés aux flot de données que ceux liés aux flot de contrôle. C'est pourquoi, il s'est avéré nécessaire dans le cadre d'AAA d'enrichir le modèle purement flot de données par des constructions de contrôle.

On rappelle que le modèle de flot de contrôle tire bien ses origines de la "machine de Von Neumann" dans laquelle, contrôlées par un compteur ordinal, les instructions sont sélectionnées pour une exécution séquentielle. Dans un tel modèle de flot de contrôle, l'accent sera mis principalement sur l'ordonnancement des opérations de calcul en définissant explicitement l'ordre d'exécution des opérations et en choisissant (par test et branchement) parmi plusieurs opérations exclusives une d'eux. Les arcs

d'un graphe flot de contrôle représentent donc un contrôle de séquencement qui correspond soit à un branchement inconditionnel (utilisé pour spécifier une itération finie ou infinie), soit à un branchement conditionnel (après évaluation d'une condition). Pour pouvoir supporter ce flot de contrôle, le modèle d'algorithme adopté dans le cadre d'AAA propose de spécifier chaque branchement arrière inconditionnel (relatif à une itération finie ou boucle) de la même manière que l'itération infinie, autrement dit sous la forme d'une répétition spatiale d'un sous-graphe, qui sera factorisé au seul motif de la répétition utilisé itérativement (transformation de la répétition spatiale factorisée en répétition temporelle), et chaque branchement conditionnel sous la forme d'une dépendance de conditionnement reliant l'opération d'évaluation de la condition d'activation à l'opération (ou le sous-graphe d'opération) conditionnée. Le conditionnement et la répétition finie dans ce modèle sont donc les équivalents, en termes de graphe de dépendances de données, des structures de contrôle respectivement le conditionnement (sélection) "If...Then...Else" et la répétition "For i=1 to N Do..." que l'on trouve dans les langages impératifs. Nous reviendrons sur ces points cruciaux (factorisation et conditionnement) plus en détail par la suite.

En conclusion, les extensions apportées à la version basique du modèle graphe flot de données ont permit d'aboutir au modèle de GFCDD (graphe factorisé et conditionné de dépendances de données) qui présente un moyen simple mais efficace d'introduire du flot de contrôle à l'intérieur du flot de données dans un même modèle de spécification. L'objectif étant de garder globalement un modèle flot de données avec ses avantages (en particulier le parallélisme pour la distribution), et en même temps d'offrir plus de flexibilité dans les spécifications des structures de contrôle (sélection : "conditionnement, exécution conditionnelle", "répétition" ...).

4.3.1 Factorisation

Comme il a été déjà souligné, le processus de factorisation consistant à réduire la taille des spécifications en factorisant les sous-graphes répétitifs à leur motif de répétition, présente des particularités différentes selon que la répétition factorisée est finie ou infinie.

Factorisation finie

Une des particularités de nos applications cibles est qu'elles intègrent souvent des algorithmes essentiellement réguliers (traitement identique pour chaque donnée) traitant généralement de très grandes quantités de données. Ceci implique des volumes de calculs suffisamment importants pour générer des répétitions de traitements identiques sur des données différentes, que l'utilisateur humain, se lassant vite des énumérations, préfère spécifier en intention sous une forme factorisée (par exemple par des points de suspension ou plus formellement par des foncteurs dans le cas des spécifications algébriques, ou encore en ne spécifiant que le graphe flot de données motif de la répétition dans le cas des graphes flot de données).

Par conséquent, la spécification de tels algorithmes, en particulier leurs parties régulières, en termes de flot de données fait souvent émerger des répétitions de motifs d'opérations (sous-graphes d'opérations identiques mais opérant sur des données différentes appelés généralement "motifs"). Cette répétition finie d'un motif peut être factorisée en remplaçant le motif répétitif par le seul motif. Ce sous-graphe motif de la répétition sera délimité, séparé du reste du graphe d'algorithme, par des sommets spéciaux appelés *sommets frontières de factorisation* dont le rôle est d'énumérer/collecter les données d'entrée et

de sortie de façon à produire les mêmes résultats que la spécification non-factorisée.

Pour fixer les idées, nous illustrons cette notion de factorisation à travers un algorithme simple de calcul de produit vectoriel de deux vecteurs V et V' (figure 4.1). Dans ce cas le sous-graphe factorisé des N (égal à 3) sous-graphes répétés représentant les multiplications des éléments respectifs des deux vecteurs est délimité par les sommets de factorisation notés F, F', J. Bien qu'à ce stade, nous n'ayons pas encore défini et étudié ces différents sommets de factorisation, nous pouvons tout de même constater qu'intuitivement les sommets F, F' de type Fork servent à produire sur leurs N sorties les N éléments des deux tableaux (V et V') de N données reçu en entrée. Chacun des éléments va être utilisé comme entrée de l'opération de multiplication (mul) dans chacune des répétitions finies du sous-graphe. Inversement, le sommet Join J qui a comme entrées les N éléments obtenus en sorties des N instances de répétition du sous-graphe motif de la répétition (i.e la multiplication (mul)), produit en sortie le tableau résultat (R) formé de ces N éléments.

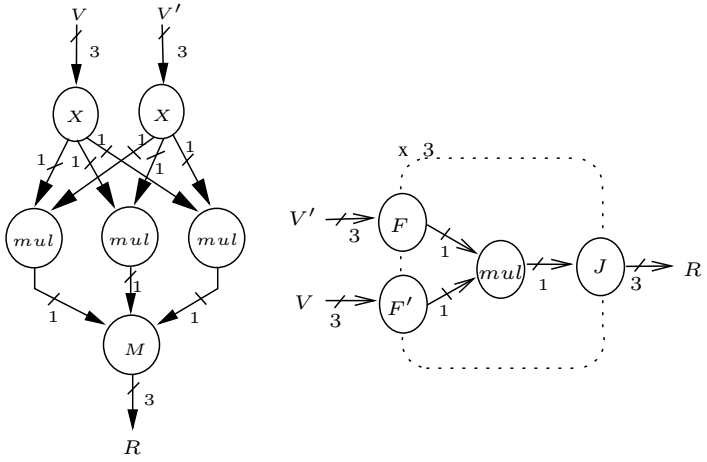


FIG. 4.1 – Exemple de graphe dépendance factorisé

Plus particulièrement, un Fork et un Join délimitant le début et la fin d'un motif factorisé correspondent au déroulement spatial d'une boucle (itération ou répétition temporelle) conduisant à du "parallélisme potentiel de données" puisque c'est le même graphe d'opérations (motif) répété spatialement qui traite des données différentes. Il est à noter aussi qu'un sous-graphe d'opérations motif d'une répétition finie, peut contenir, à son tour, un sous-graphe lui aussi répété un nombre fini de fois correspondant ainsi à des "nids de boucles". L'imbrication des boucles conduit ainsi à de la hiérarchie de sous-graphes dans le graphe d'algorithme, ce qui se traduit par de l'imbrication aux niveaux des frontières de factorisation qui délimitent les motifs répétitifs respectifs.

Il est clair que la factorisation n'a pas pour seul intérêt de réduire la taille des spécifications sans changer en rien l'ordre partiel entre les opérations du graphe : elle permet aussi de réduire en intention la taille des architectures obtenues par traduction directe du graphe de dépendances factorisé (implantation séquentielle de la répétition temporelle). Une opération située à l'intérieur du sous graphe factorisé d'opérations identiques opérant sur un ensemble factorisé de données différentes, se traduit directement par un seul opérateur (ou de plusieurs opérateurs en parallèle) utilisé(s) autant de fois qu'il

existe d'opérations dans le groupe factorisé. Chaque ensemble factorisé de données doit donc être multiplexé : alors qu'il ne joue qu'un rôle "syntaxique" au niveau du graphe de dépendances, les opérateurs correspondant aux sommets frontières doivent réaliser le multiplexage.

On comprend donc aisément que l'on peut transformer une répétition spatiale en répétition temporelle, en transformant la répétition spatiale en une itération (transformation spatiale-temporelle), ce qui diminue les ressources nécessaires lors de l'implantation, et inversement on peut transformer une itération en une répétition spatiale si les ressources le permettent. Ainsi, la répétition spatiale factorisée peut être inversement transformée en répétition temporelle et vice versa si cela est nécessaire lors des optimisations.

De ce fait, la factorisation d'un motif répétitif ne présente pas seulement un équivalent à la structure de boucle dans un langage impératif, qui impose un ordre total sur les itérations, elle offre une sémantique plus forte via les possibilités de transformations spatiales-temporelles qui autorisent aussi bien un ordre partiel qu'un ordre total entre les itérations.

L'intérêt de la factorisation ne se restreint ainsi pas uniquement à réduire la taille de la spécification algorithmique en mettant en évidence ses parties régulières (motifs), sans en modifier la sémantique opératoire, elle permet aussi de décrire en intention plusieurs implantations plus ou moins séquentielles ou parallèles, chacune avec des caractéristiques (surface, temps de réponse) différentes.

Sommets de factorisation Cette opération de factorisation fait apparaître des sommets spéciaux (sommets frontières de factorisation), qui servent à délimiter et mettre en évidence le sous-graphe motif de la factorisation, ainsi qu'à spécifier l'une des différentes manières de factoriser les données et les opérations en traversant la frontière du motif factorisé :

Sommet Fork Noté 'F', il effectue la factorisation d'un flot de données en partitionnant et distribuant la donnée en entrée aux différents motifs factorisés en sortie ; c'est-à-dire si SG_i avec $1 \leq i \leq n$ sont les n instances du sous-graphe à factoriser, le sommet de factorisation Fork divise la dimension d de la donnée en son entrée en d/n éléments qu'il énumère un à un en sortie (partition d'un tableau en autant d'éléments que de répétitions du motif SG_i sur la figure 4.2). Dans le cas d'une implantation séquentielle répétitive de la factorisation, le sommet Fork permet de traverser une frontière de factorisation, tout en augmentant la cadence des données, en aval la fréquence des données est n fois supérieure à la cadence des données en amont. Ainsi l'entrée du Fork est du coté lent de la frontière qu'il délimite alors que ses sorties sont du coté rapide.

Sommet Join Noté 'J', il effectue la factorisation des flots de données en son entrée en les collectionnant et regroupant en un vecteur qu'il fournit en sortie ; c'est-à-dire si SG_i avec $1 \leq i \leq n$ sont les n instances des sous graphes à factoriser, le sommet de factorisation Join regroupe les n données de dimension d issues des ces n motifs en un même vecteur de données de dimension $n * d$ (composition d'un tableau à partir des résultats de chaque répétition du motif SG_i sur la figure 4.3). Dans le cas d'une implantation séquentielle répétitive, le sommet Join permet donc de traverser une frontière de factorisation, tout en réduisant la cadence des données en amont la fréquence des données est n fois supérieures à la cadence des données en aval. Ainsi l'entrée du Join est du coté rapide de la frontière qu'il délimite alors que sa sortie est du coté lent.

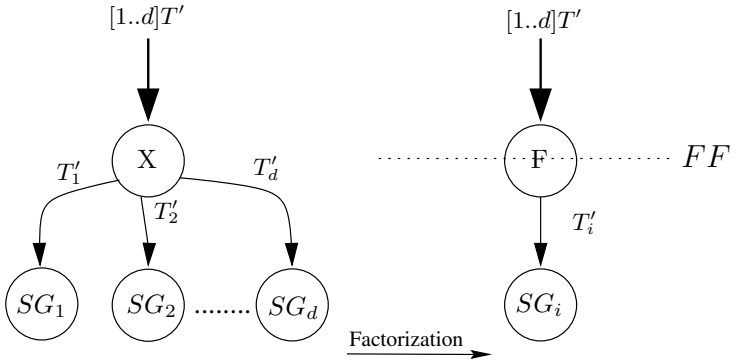


FIG. 4.2 – Sommet de factorisation Fork

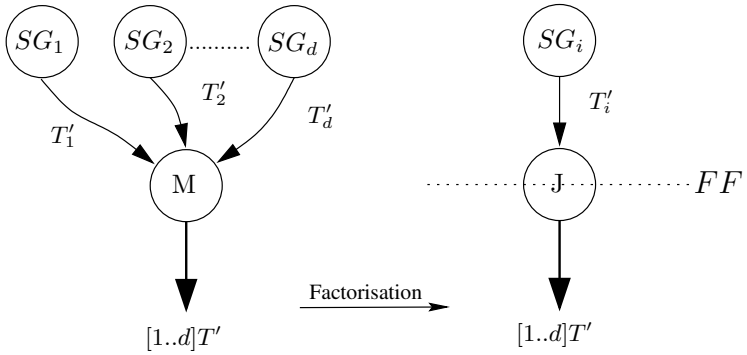


FIG. 4.3 – Sommet de factorisation Join

Sommet Diffuse Noté 'D', il effectue la factorisation d'une seule donnée utilisée n fois par le motif répétitif ; c'est-à-dire si SG_i avec $1 \leq i \leq n$ sont les n instances des sous-graphes à factoriser, la donnée en entrée du sommet de factorisation Diffuse sera diffusée aux différents motifs factorisés SG_i en sa sortie (diffusion d'une donnée à toutes les répétitions du motif SG_i sur la figure 4.4). Dans le cas d'une implantation séquentielle répétitive, le sommet D ne modifie pas le flot de données, le diffusant tel qu'il se présente du coté lent de la frontière vers le côté rapide. Il ne sert qu'à marquer la frontière de factorisation au niveau d'un arc qui la traverse.

Sommet Iterate Noté 'I', il effectue la factorisation des dépendances de données inter-motifs. Lorsque ces n sous graphes SG_i (avec $1 \leq i \leq n$) sont factorisés, chaque dépendance de données inter-motif apparaît en sortie et en entrée du sous-graphe factorisé (dépendance de donnée inter-itération du motif). Ainsi la connexion inter-motifs apparaît dans le graphe factorisé comme un cycle à travers le sommet ITERATE. Ce sommet marque une dépendance de donnée inter-répétition, avec une seconde entrée "initiale" pour la première répétition, et une seconde sortie "finale" pour la dernière répétition (voir figure 4.5). Dans le cas d'une implantation séquentielle répétitive la cadence des données intermédiaires (i_i et o_i) est n fois supérieure à la cadence des flots de données "initial" ($init$) et "final" (fin). Les flots de données $init$ et $final$ sont du coté lent de la frontière de factorisation et les flots intermédiaires

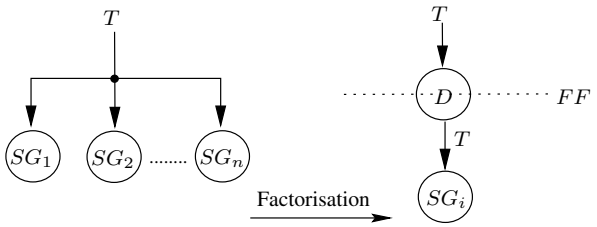


FIG. 4.4 – Sommet de factorisation Diffuse

d'entrée et de sortie i_i et o_i sont du coté rapide.

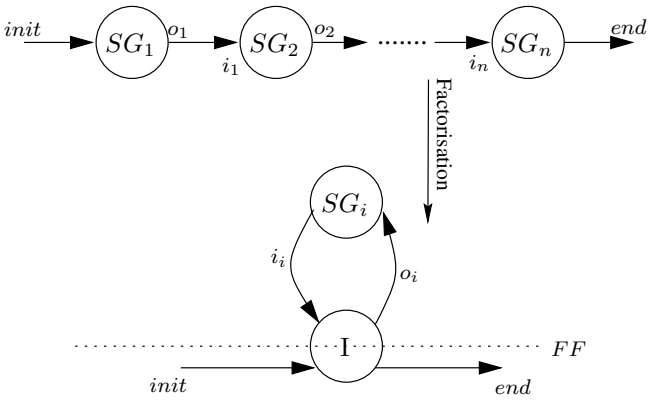


FIG. 4.5 – Sommet de factorisation Iterate

Notons que la définition d'un tel sommet de factorisation apporte une amélioration intéressante au modèle graphe flot de données classique. Puisque normalement son formalisme de flot de données n'est pas approprié pour passer des données d'une boucle d'itération à la prochaine dans les constructions d'itération, le sommet iterate permet de remédier à cette insuffisance en autorisant le passage de donnée d'une itération à une autre.

Frontières de factorisation C'est une abstraction délimitée par un ou plusieurs sommets particuliers de factorisation qui servent à spécifier le début ou la fin d'une factorisation.

Une frontière englobera tous les traitements qui doivent être répétés un certain nombre de fois (appelé facteur de factorisation) pour exécuter le motif correspondant, elle ne fait que délimiter deux parties du graphe factorisé ayant des facteurs de répétition différents et par conséquent appartenant à des motifs différents.

Notons que dans le cas d'une implantation séquentielle (transformation de la répétition temporelle en itération), l'une des deux parties est dite "rapide" multiple de l'autre "lente". Pendant que les opérateurs du côté lent sont exécutés une fois, ceux du côté rapide seront exécutés plusieurs fois. Ce rapport de factorisation entre côté lent et côté rapide, appelé facteur de factorisation permet, selon son degré, de spécifier les opérations d'un algorithme sous différentes formes plus ou moins factorisées. La détermination de ce rapport peut être faite en examinant la taille des vecteurs d'entrée et de sortie des

sommets frontières de type Fork et Join du même motif :

$$n = \left\{ \frac{dimension\ du\ vecteur\ d'entrée}{dimension\ du\ vecteur\ de\ sortie} \right\}_{Fork} = \left\{ \frac{dimension\ du\ vecteur\ de\ sortie}{dimension\ du\ vecteur\ d'entrée} \right\}_{Join}$$

Lors du passage au travers d'une frontière de factorisation, un vecteur d'entrée peut être soit distribué tel quel n fois en sortie (sommet Diffuse), soit partitionné en n sous-vecteurs de sortie (sommet Fork), n vecteurs d'entrée peuvent être soit regroupés en un vecteur de sortie (sommet Join), soit être combinés entre eux (sommet Iterate). Ainsi à chaque itération :

- les données en entrée d'une frontière proviennent soit :
  - d'un vecteur : un scalaire ou sous-vecteur à chaque itération (cas du Fork),
  - toujours de la même source (cas du Diffuse),
- les données en Sortie d'une frontière :
  - regroupent les résultats des itérations (cas du Join),
  - repésentent la valeur de la dernière itération dans le cas où il existerait des dépendances inter-itération (cas de l'Iterate).

Factorisation infinie et sommet retard

Comme on s'intéresse à spécifier des systèmes temps réel réactifs qui interagissent avec leur environnement d'une manière discrète, sous forme d'une répétition infinie de la séquence acquisition-calculs-commande, on a besoin de spécifier cette interaction répétitive avec l'environnement. Cette répétition présente pour particularités d'une part d'être infinie et d'autre part de contraindre les données d'entrée et les résultats de sortie à être multiplexés (le plus souvent périodiquement) à travers les capteurs et actionneurs réalisant l'interface avec l'environnement.

Pour ce faire, on définit par analogie à la factorisation finie, une nouvelle version de chaque sommet de factorisation finie Fork, Join, Iterate et Diffuse qui se chargeront de la spécification de la factorisation infinie du graphe de dépendances infiniment répétitif modélisant l'interaction infinie avec l'environnement. Les sommets de factorisation infinie $Fork^{\infty}$, $Join^{\infty}$, $Iterate^{\infty}$ et $Diffuse^{\infty}$ modélisent donc l'interface avec l'environnement physique des systèmes réactifs et sont alors respectivement équivalents aux entrées acquises par les capteurs, et aux sorties produites par les actionneurs, les retards inter-répétition et les constantes du graphe de flots de données. Le sommet particulier $Itérate^{\infty}$ appelé aussi "retard" permet d'implanter les dépendances de données inter-répétition et offre de ce fait un mécanisme de mémoire ou d'état permettant au sein d'une interaction d'accéder au passé des précédentes interactions (équivalent au z^{-1} du traitement du signal). Il stocke l'état de l'algorithme qu'il est important de bien maîtriser en particulier dans les algorithmes de contrôle-commande afin de leur assurer de bonnes propriétés (stabilité, commandabilité, observabilité).

L'exemple de la figure 4.6 présente un graphe de dépendances répétitif de taille infinie (points de suspension) et sa factorisation. À chaque répétition $t = i$, l'opération de calcul D acquiert une nouvelle entrée par l'intermédiaire du capteur C_2, et l'opération B acquiert une nouvelle entrée par l'intermédiaire du capteur C_1, et la combine avec l'entrée dont la valeur est égale à la valeur calculée par D lors de l'répétition précédente $t = i - 1$. Le sommet retard noté \$, permet de fournir la sortie D_t en entrée à l'opération B lors de la réaction suivante $t + 1$.

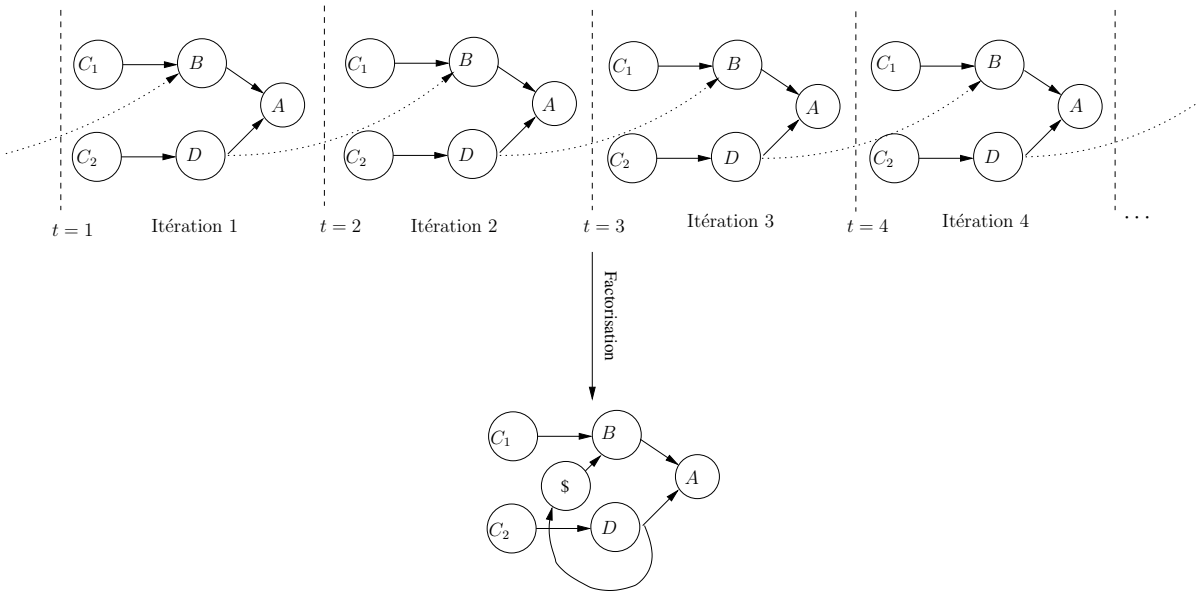


FIG. 4.6 – Graphe flot de données infiniment factorisé et sommet retard

Enfin, nous considérons que la spécification d'un algorithme est toujours implicitement factorisée, les sommets Fork$^\infty$ (capteurs) et Join$^\infty$ (actionneurs) réalisent matériellement le multiplexage des données en entrée et en sortie.

4.3.2 Conditionnement

L'exécution conditionnelle est le complément en termes de structures de contrôle à la répétition (boucle) en vue de permettre un support complet du flot de contrôle. L'introduction d'une telle structure est donc indispensable à l'enrichissement du modèle de flot de données pur afin d'offrir une modélisation complète des algorithmes spécifiés. C'est l'une des caractéristiques qu'un modèle de spécification basé sur le flot de données nécessite pour être puissant.

Compte tenue du fait qu'historiquement, la spécification algorithmique utilisée dans la méthodologie AAA était très proche de celle du langage synchrone Signal, la spécification du conditionnement adoptée sera basée sur la notion de condition d'activation tel qu'il est le cas dans le format commun des langages déclaratifs synchrones, DC ((pour "Declarative Code" en anglais) [123]. Ce qui permettra, de simplifier d'une part la spécification du conditionnement, et d'autre part d'élargir la compatibilité du modèle à tous les langages synchrones. Dans un modèle basé sur la notion de condition d'activation, on ne "contrôlera" pas directement les flots mais on "conditionnera" une opération particulière par l'intermédiaire d'une dépendance d'entrée spécifique appelée "condition d'activation" en plus des autres dépendances d'entrée et de sortie de l'opération. Le flot de contrôle sera ainsi géré par des instructions de branchement conditionnel dont la condition est extraite du booléen de la condition d'activation.

Dès lors, la modélisation du conditionnement au niveau du graphe d'algorithme nécessite l'introduction d'hyperarcs et de sommets spécifiques appelés respectivement arc de dépendances de donnée de conditionnement et sommet "select".

Arc de dépendance de donnée de conditionnement

Chaque condition d'activation est modélisée par un arc de dépendance de donnée de conditionnement appelé arc de conditionnement [32] (différent des dépendances de données) qui induit une condition d'activation sur chaque opération réceptrice d'un tel arc. Les opérations puits de ces arcs sont dites *conditionnées*, elles ne sont exécutables que si leur condition d'activation est présente et transmet la valeur vraie (i.e valeur du booléen correspondant) et que leurs autres données sont présentes. De ce fait, bien qu'on est dans un modèle flot de données, la prise en compte du contrôle dû au conditionnement exige qu'une opération conditionnée ne peut exécuter son calcul et produire des données en sortie que lorsque sa condition d'activation est présente et transmet la valeur vraie même si l'ensemble des autres données en entrée est présent. Cette règle d'activation est bien conforme à la manière dont le conditionnement est traité dans la version flot de données de Denis [44, 43].

Cette introduction du contrôle lié au conditionnement entraîne donc l'émergence de deux types de dépendances de données entre les opérations : les dépendances de données de conditionnement, et les dépendances de données. De ce fait, le modèle d'algorithme "GFCDD" constituera un type spécial de graphe orienté, où les dépendances de données et de contrôle (lié au conditionnement) sont représentées de manière uniforme, orientée flot de données sous forme de dépendances de données. Pour les différencier graphiquement au niveau du graphe d'algorithme, nous représenterons les dépendances de données de conditionnement par des arcs en pointillés et ceux de données par des arcs en traits continus. Notons que ces arcs de dépendances de conditionnement transmettent le booléen de la condition d'activation aux opérations dites 'conditionnées' d'où l'appellation de *"conditionnement par booléen"*.

Sommet Select

Par analogie au EndIF (End Case, ..) des structures de contrôle conditionnelles (IF, CASE,..) dans la spécification séquentielle faite avec les langages impératifs, nous avons défini un sommet opération spécifique 'select' qui permet de délimiter le sous-graphe d'opérations conditionnées dans le graphe d'algorithme et de sélectionner parmi les données reçues en entrée ($D_1, D_2, ..., D_n$ sur la figure 4.7) celle qui sera acheminée en sortie (sélection de données). Ces entrées correspondent aux données produites par les opérations conditionnées ayant leur condition d'activation vérifiée. Plusieurs cas de figures peuvent ainsi se présenter en entrée de ce sommet selon si la condition d'activation est vérifiée ou non et conditionnent par conséquent son fonctionnement.

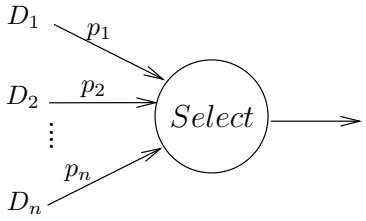


FIG. 4.7 – Sommet select

Sélection des données en entrée L'exécution parallèle des opérations conditionnées non nécessairement exclusives peut conduire à la présence simultanée de plusieurs données en entrée du sommet select, nous avons donc introduit un ordre de priorité entre ces données qui sera spécifié de manière explicite avec des étiquettes p_i sur ses arcs d'entrées comme illustré sur la figure 4.7. L'entrée présente ayant la plus haute priorité correspond à l'entrée qui sera sélectionnée et dont la valeur sera acheminée en sortie du sommet select.

Rupture du flot Comme nous sommes dans une approche flot de données, notre modèle de spécification étendu au contrôle doit apporter une attention particulière à la continuité du flot. En effet, les branchements conditionnels provoquent généralement des ruptures de flot de données. Ce problème de rupture du flot dû au conditionnement est l'un des principaux goulots d'étranglement retardant la prise en compte du contrôle conditionnel dans les approches flots de données.

Plus précisément, lorsqu'aucune condition n'est vérifiée, le problème qui se pose est celui de la valeur de donnée que le "Select" doit fournir en sortie à ses successeurs. Pour résoudre ce problème plusieurs approches ont été adoptées dans la littérature :

Format commun des langages synchrones DC Le format DC (pour "Declarative Code" en anglais), est le format intermédiaire utilisé par les compilateurs des langages synchrones (Esterel, Lustre et Signal pour l'instant). Dans ce format, si aucune des conditions conditionnant les entrées du sommet "select" n'est vérifiée, le flot :

- est non défini si la condition n'a jamais eu lieu avant et qu'il n'y a pas de valeur donnée par défaut,
- prend la valeur définie par défaut si elle existe et si la condition n'a encore jamais été vraie,
- prend la dernière valeur définie quand sa condition d'activation était vraie.

Modèle Signal Le langage déclaratif synchrone Signal procède à l'arrêt du flot si aucune condition n'est vérifiée.

Dans notre modèle pour résoudre ce problème de rupture de flot, nous proposons d'ajouter une opération *default* (valeur par défaut) à l'entrée du sommet Select, dont le rôle est de recopier en sortie une valeur constante qui sera acheminée en entrée du sommet Select via un arc étiqueté par la plus faible priorité p_i. Ce qui permettra le cas échéant d'aiguiller cette valeur en sortie du sommet Select. Cette approche de résolution du problème présente une solution intermédiaire entre les deux approches citées ci-dessus, d'une part elle assure la continuité du flot et ne procède pas à son arrêt et d'autre part elle n'exige pas une prise en compte complexe nécessitant de la mémorisation.

Nous illustrons de manière assez simple notre modèle de graphe GFCDD à travers l'exemple de la figure 4.13 qui calcule selon la valeur d'une entrée x l'addition des deux entrées e_1 et e_2 :

Remarque 1 *Notons que pour les structures de contrôles conditionnelles consistant en des comparaisons avec des valeurs constantes, la spécification du conditionnement peut être simplifiée, en étiquetant les arcs de dépendances de données de conditionnement par la condition portant sur la valeur transmise aux opérations conditionnées. Nous appelons ce mode de conditionnement 'conditionnement non booléen'. Du fait qu'on ne spécifie pas explicitement sur le graphe le sommet calculant la condition ('cond' sur la règle citée ci-dessous) le nombre de sommets des graphes issus d'une telle modélisation du conditionnement basée sur le conditionnement non booléen sera visiblement réduit par rapport à ceux*

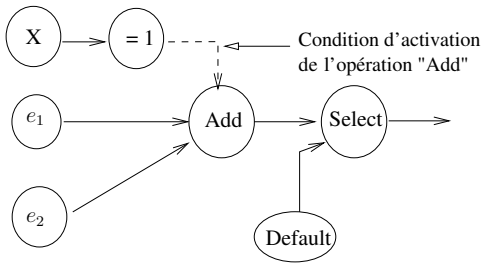


FIG. 4.8 – Exemple de graphe conditionnée

issus d'une modélisation basée sur un conditionnement booléen. Ce qui améliore considérablement la lisibilité dès que l'algorithme spécifié contient un nombre important de comparaisons avec des constantes et d'autre part réduit le nombre de sommets à traiter par les transformations en vue de l'implantation.

Propriété 1 *Pour ces structures de contrôle particulières, on peut définir une transformation simple permettant le passage d'une spécification du conditionnement par booléen en une spécification du conditionnement non booléen comme suit : pour tout sous-graphe linéaire composé de la condition (opération calculant la condition)* $Cond_i$ *suivie de l'opération conditionnée* O_j *sera remplacé par une dépendance de donnée et de conditionnement* $d_{i_{dc}}$ *représentée par un arc en pointillé étiqueté par la condition* $Cond_i$ *portant sur l'entier transmis (voir figure 4.9).*

$$(O_i, \underbrace{d_{i_d}, Cond_i, d_{i_c}}_{d_{i_{dc}}}, O_j) \overset{Transformation}{\rightarrow} (O_i, d_{i_{dc}}, O_j)$$

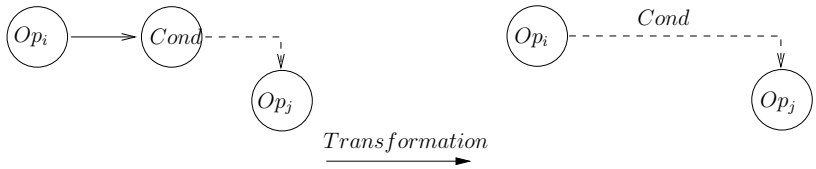


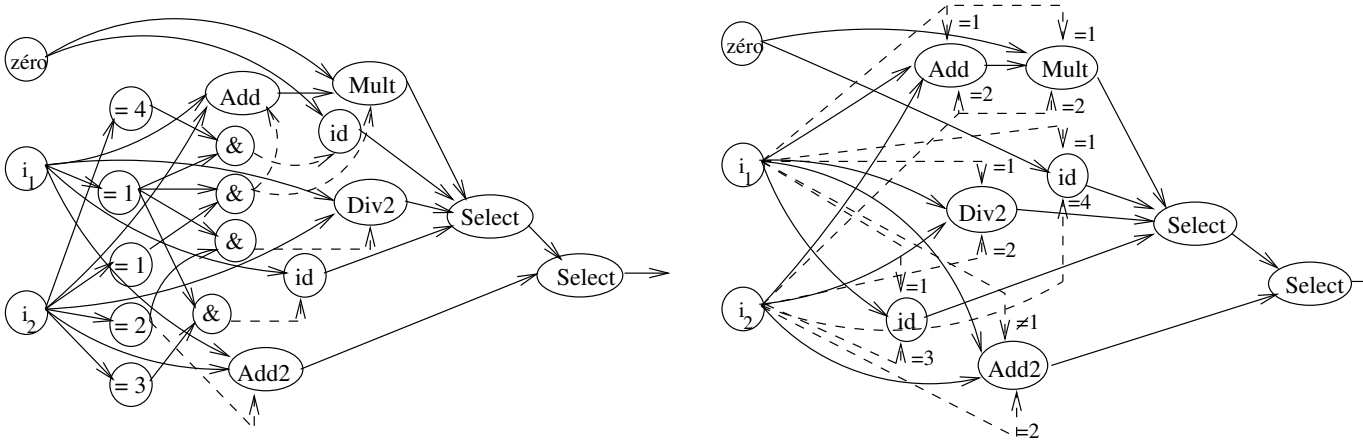
FIG. 4.9 – Règle de transformation

Par la suite quand on parlera de conditionnement on sous-entendra implicitement une spécification du conditionnement par booléen puisque c'est la modélisation qui couvre un large spectre des structures conditionnelles.

Exemple 1 La figure 4.10 correspondant à une modélisation graphique de l'algorithme suivant, montre le gain en nombre de sommets et en lisibilité lorsqu'on adopte une spécification du conditionnement basée sur le conditionnement non booléen pour des structures conditionnelles présentant la particularité de comparaisons à des constantes.

Begin
if i_1=1 **then**
  if i_2=1 **then**

$mulimb_{in}$:= Addimb (i_1, i_2) ;
o := Mulimb($addimb_{out}$, $Const_{zero}$) ;
else
if i_2=2 **then**
o := div2(i_1, $Const_{zero}$) ;
else
if i_2=3 **then**
o := i_1
else
if i_2=4 **then**
o := $Const_{zero}$;
end if
end if
end if
end if
else
if i_2=2 **then**
o := Add2(i_1, i_2) ;
end if
end if
End



a) Conditionnement par booléen

b) Conditionnement par non booléen

FIG. 4.10 – Exemple de graphes conditionnés

4.3.3 Exemple de spécification algorithmique

Nous allons illustrer le modèle de graphe factorisé et conditionné des dépendances de données (GFCDD) présenté dans ce chapitre par l'algorithme suivant qui selon la valeur de C en entrée calcule,

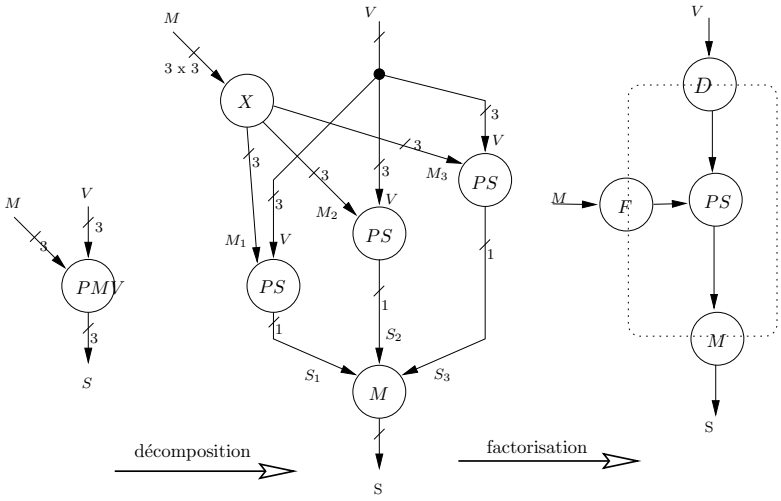


FIG. 4.11 – Décomposition/factorisation du produit matrice-vecteur

soit le produit d'une matrice $M \in R^m \times R^n$ par un vecteur $V \in R^n$ et fournit en sortie le vecteur $S \in R^m$ résultant, soit fournit directement en sortie le vecteur V d'entrée. Le choix de cet exemple est dû d'une part à son intérêt pratique : le produit matriciel est un exemple de traitement très fréquent en applications de traitement d'image et d'autre part pour sa simplicité. Il exprime de manière claire et simple les motifs répétitifs, la notion de factorisation ainsi que le support des structures de contrôle de répétition et de sélection (For et IF).

```
Begin
if C=1 then
  for i=1 to m do
    for j=1 to n do
      S[i] := S[i]+M[i,j]*V[j] ;
    end for
  end for
else
  S := V ;
end if
End
```

Le calcul du produit de la matrice M (composée de m vecteurs M_i : $M = (M_i)_{1 \leq i \leq m}$) par le vecteur V peut se décomposer en m produits scalaires $PS = (M_i V)_{1 \leq i \leq m}$ (loop **for i**) qui peuvent se décomposer chacun en une somme de n produits $M_i V = M_{i1}V_1 + M_{i2}V_2 + ... + M_{in}V_n$ (loop **for j**). Comme il a été mentionné précédemment cette décomposition en opérations implémentables (multiplication, addition) génère des répétitions de motifs d'opérations que l'on préfère souvent spécifier sous forme factorisée (figures 4.11 et 4.12). Sur la figure 4.11 on présente le graphe de dépendances du produit-matrice vecteur pour $m = n = 3$ (à gauche), sa décomposition en produits scalaires (au centre), et sa factorisation (à droite) qui fait apparaître trois sommets spéciaux de factorisation (D, F, J) qui délimitent la frontière, mise en évidence par des pointillés.

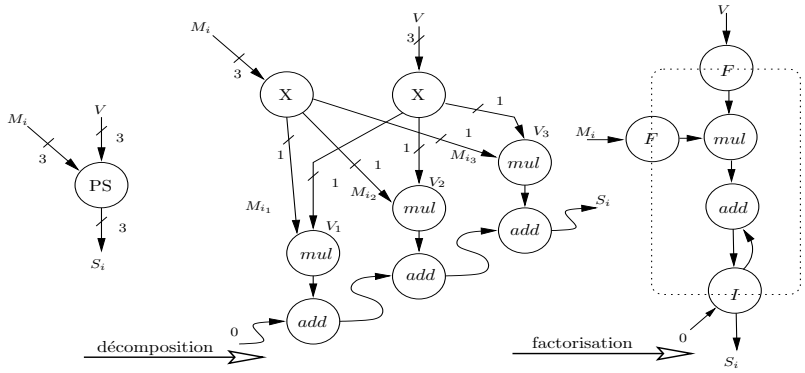


FIG. 4.12 – Décomposition/factorisation du produit scalaire

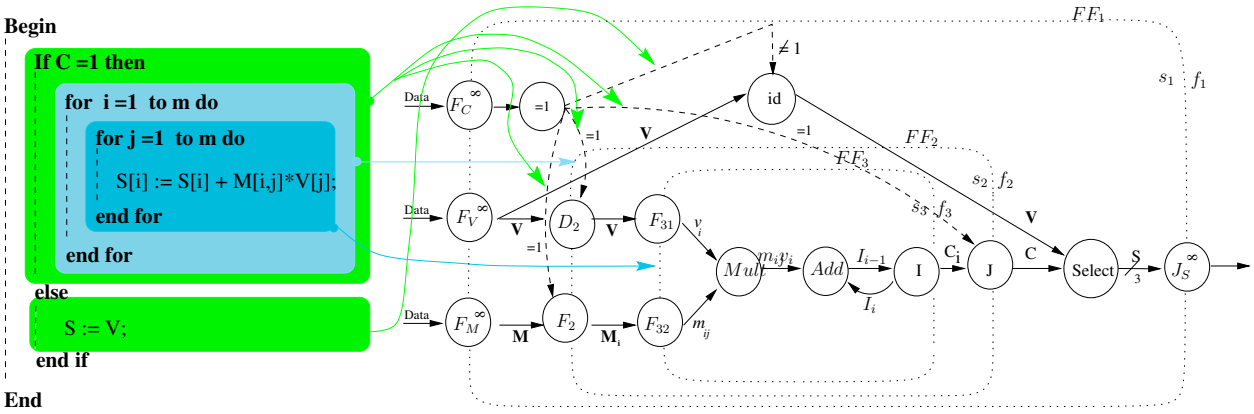


FIG. 4.13 – Le GFCDD de l'algorithme de l'exemple

La figure 4.12 présente le graphe de dépendances du produit scalaire, sa décomposition en somme de produits et sa factorisation (à droite) qui fait apparaître un autre sommet frontière I.

Par conséquent, le Graphe Factorisé de Dépendance de Données correspondant à la spécification séquentielle algorithmique de départ (Fig.4.13) présente deux frontières imbriquées FF_2 et FF_3 chacune d'elle correspond à un niveau de factorisation des motifs répétitifs des calculs contenus dans la boucle FOR correspondante. La frontière extérieure FF_1 représente l'interface avec l'environnement et correspond à la factorisation du motif de graphes répétitifs infinis de la répétition spatiale infinie du modèle flot de données. Vu que la fréquence de consommations et productions de données diffère au niveau de chaque côté des zones délimitées par une frontière de factorisation FF. On désigne ainsi par "rapide"(*fast 'f'*) le côté de la frontière étant répété le plus par rapport à l'autre "lente"(*slow 's'*). Par exemple, à chaque consommation d'un vecteur V et d'une ligne M_i par la frontière FF_3 les opérations de calculs *mul* et *add* consomment et produisent des données autant de fois qu'il y a d'éléments dans V (M). Ainsi *mul* et *add* situés à l'intérieur de FF_3 définissent le côté rapide de FF_3 noté par f_3 et par conséquent son facteur de répétition (ici 3).

La structure conditionnelle IF THEN ELSE est modélisé au niveau du graphe d'algorithme par des arcs en pointillés transportant les conditions d'activation et par le sommet select permettant d'acheminer en sortie le résultat du calcul effectué.

Comme le calcul effectué dans la structure répétitive FOR (loop **for i**) est conditionné, la dépendance de conditionnement porte donc sur tous les sommets frontières de la frontière FF_2 correspondante.

4.3.4 Formalisation

Hypergraphe orienté

L'algorithme, modélisé par un graphe de dépendances de données factorisé conditionné (GFCDD), est un hypergraphe orienté G_{al} dont les sommets sont des opérations de calculs, de factorisation, de conditionnement ou d'entrée-sortie et les arcs sont soit des dépendances de données ou de conditionnement. Dès lors, le graphe d'algorithme G_{al} est un couple (O, D) où :

- O est l'ensemble fini des opérations de G_{al} : $O = \{o_i\}_{1 \leq i \leq n}$ avec Card $O = n$.
- D est l'ensemble fini des arcs, appelés dépendances de données inter-opérations $D \subseteq O \times P(O)$ $D = \{d_i\}$ tel que : $d_i = (o_{i_1}, \{o_{i_j}\}_{2 \leq j \leq k(i) \leq n)})o_{i_j}$ tous différents.

Associé à l'ensemble des dépendances D, on définit la fonction γ^{-1} qui, à chaque dépendance d_i, associe son opération émettrice $\gamma^{-1}(d_i)$ (appelée aussi productrice) et on définit la fonction γ qui, à chaque dépendance d_i, associe l'ensemble des opérations réceptrices (appelées aussi consommatrices).

$$\begin{aligned} \gamma^{-1} : \quad & D \longrightarrow O \\ & d_i \longmapsto \gamma^{-1}(d_i) = (o_{i_1}) \text{ où } d_i = (o_{i_1}, \{o_{i_j}\}_{2 \leq j \leq k(i) \leq n}) \\ \gamma : \quad & D \longrightarrow P(O) \\ & d_i \longmapsto \gamma(d_i) = (o_{i_j}) \text{ où } d_i = (o_{i_1}, \{o_{i_j}\}_{2 \leq j \leq k(i) \leq n}) \end{aligned}$$

Propriété 2 *Le graphe est sans circuit. Il n'existe pas de chemin ayant à la fois même origine et même destination. Chaque dépendance de données a un et un seul émetteur et au moins un récepteur, ainsi :*

$$\forall d_i \in D \; Card(\gamma^{-1}(d_i)) = 1 \; et \; Card(\gamma(d_i)) \geq 1$$

Remarque 2 *Notons que quand* $Card(\gamma(d_i)) \geq 1$,d_i *est un hyperarc possédant un émetteur et plusieurs récepteurs, on dit que l'on a de la diffusion,* d_i *diffuse ses données.*

Relations entre opérations Associé au graphe G_{al}, on définit les notions de successeur et de descendant, ainsi que de prédécesseur et d'ancêtre d'une opération.

- Successeurs d'une opération au sens de Gondran et Minoux : $\Gamma(o_i)$
  Soit $\Gamma(o_i)$, l'ensemble des successeurs d'une opération o_i.

$$\Gamma(o_i) = \{o_j \in O \; / \; \exists d_k \in D \; avec \; o_i = \gamma^{-1}(d_k) \; et \; o_j \subseteq \gamma(d_k)\}$$

  Les opérations sans successeur du graphe G_{al} sont appelées les sorties.

$$Sorties(G_{al}) = \{o_i \in O \; / \; \Gamma(o_i) = \phi\}$$

– Descendants d'une opération au sens de Gondran et Minoux : On suppose que l'opération o_i est au maximum à n arcs d'un noeud sans successeur. Soit l'ensemble des descendants d'une opération o_i. C'est la fermeture transitive de l'ensemble des successeurs, c'est-à-dire :

$$\hat{\Gamma}(o_i) = \bigcup_{k=1}^{n} \Gamma^k(o_i)$$

où $\Gamma^k(o_i)$ désigne l'ensemble des opérations que o_i atteint en utilisant exactement k arcs.

– Prédécesseurs d'une opération au sens de Gondran et Minoux : $\Gamma^{-1}(o_i)$
Soit $\Gamma^{-1}(o_i)$, l'ensemble des prédécesseurs d'une opération o_i.

$$\Gamma^{-1}(o_i) = \left\{ o_j \in O \ / \ \exists d_k \in D \ avec \ o_j = \gamma^{-1}(d_k) \ et \ o_i \subseteq \gamma(d_k) \right\}$$

Les opérations sans prédécesseur du graphe G_{al} sont appelées les entrées.

$$Entrées(G_{al}) = \left\{ o_i \in O \ / \ \Gamma^{-1}(o_i) = \phi \right\}$$

– Ancêtres d'une opération au sens Gondran & Minoux :
On suppose que l'opération o_i est à exactement n arcs d'un noeud sans prédécesseur. Soit l'ensemble des ancêtres d'une opération o_i. C'est la fermeture transitive de l'ensemble des prédécesseurs, c'est-à-dire :

$$\hat{\Gamma}^{-1}(o_i) = \bigcup_{k=1}^{n} (\Gamma^{-1})^k(o_i)$$

où $(\Gamma^{-1})^k(o_i)$ désigne l'ensemble des opérations atteignant en utilisant exactement k arcs.

Hypergraphe conditionné

Dans le graphe $G_{al} = (O, D)$ déjà défini
– l'ensemble fini des opérations O se décompose en deux sous-ensembles disjoints :
  – O_{ncond} : ensemble des opérations non conditionnées, c'est-à-dire qui s'exécutent sans condition,
  – O_{cond} : ensemble des opérations conditionnées, l'exécution dépend d'une condition d'activation.
– l'ensemble fini des dépendances D se décompose en deux sous-ensembles disjoints :
  – D_d ensemble des dépendances de données dites *dépendances de données simples*,
  – D_c ensemble des dépendances de données et de conditionnement, appelées *dépendances de conditionnement*.

Propriété 3 – *Les ensembles O_{ncond} et O_{cond} forment une partition de l'ensemble des sommets O ($O_{ncond} \neq \phi$ et $O_{cond} \neq \phi$, $O_{ncond} \cap O_{cond} = \phi$, $O_{ncond} \cup O_{cond} = O$).*
– *Les ensembles D_d et D_c forment une partition de l'ensemble des arcs D ($D_d \neq \phi$ et $D_c \neq \phi$, $D_d \cap D_c = \phi$, $D_d \cup D_c = D$).*

Hypergraphe factorisé

Les sommets $o_i \in O$ du graphe d'algorithme $G_{al} = (O, D)$ seront étiquetés avec des informations concernant leurs frontières d'appartenance que nous notons par :

- o_i^{c,FF_i} pour toute opération $o_i \in O$ se situant du coté c de la frontière FF avec $c = s$ pour lent ($s$low), f pour rapide ($f$ast), ou s/f si o_i est une opération de factorisation de FF.

- ou simplement $o_i^{FF_i}$ pour toute opération $o_i \in O$ appartenant à FF_i (opération de factorisation de FF_i "$c = f/s$" ou opération de calcul directement incluse dans FF_i ($c = s$ i.e de son côté lent)).

Définition 1 *Soit $\Re$ la relation d'appartenance à une même frontière de factorisation définie sur l'ensemble des opérations (sommets) O comme suit :*

$$o_i^{FF_k} \Re o_j^{FF_l} \Leftrightarrow k = l$$

Propriété 4 *$\Re$ est une relation d'équivalence sur O, elle réflexive, symétrique et transitive.*

Propriété 5 *Les classes d'équivalence $O/\Re$, noté $\dot{F}_i$, définissent l'ensemble des frontières de factorisation FF_i du graphe factorisé G_{al} noté F.*

Associé à l'ensemble des sommets O, on définit la fonction ψ qui, à chaque sommet opération o_i, associe sa frontière de factorisation $\psi(o_i) = FF_i$ et on définit la fonction ψ^{-1} qui, à chaque frontière de factorisation FF_i, associe l'ensemble de ses opérations.

$$\begin{aligned} \psi : \quad & O \longrightarrow F \\ & o_j^{FF_i} \longmapsto \psi(o_j^{FF_i}) = FF_i \text{ où } FF_i \text{ est la } frontière \text{ de } factorisation \text{ de } o_j \end{aligned}$$

$$\begin{aligned} \psi^{-1} : \quad & F \longrightarrow P(O) \\ & FF_i \longmapsto \psi^{-1}(FF_i) = \left\{ o_j^{FF_l} \right\} \text{ où } FF_i = FF_l \end{aligned}$$

Comme chaque frontière FF_i définit un rapport de factorisation entre son côté lent s "$s$low" et son côté rapide f "$f$ast", appelé $f$acteur de factorisation, on associe à l'ensemble des frontières F ($F = O/\Re$) la fonction φ qui, à chaque frontière FF_i associe son facteur de factorisation $fact$.

$$\begin{aligned} \varphi : \quad & F \longrightarrow N \\ & FF_i \longmapsto \varphi(FF_i) = fact_i \text{ où } fact_i \text{ est le } facteur \text{ de } factorisation \text{ de } FF_i. \end{aligned}$$

On définie de manière annalogue la fonction ϕ qui, à chaque frontière $FF_i \in F$ associe son facteur de défactorisation $defact$.

Étiquetage des arcs

Le formalisme présenté jusqu'ici pour modéliser les algorithmes, est suffisant pour décrire les algorithmes ainsi que leurs propriétés en termes de graphe. Cependant, comme dans cette thèse l'objectif recherché est de formaliser tout le processus d'implantation depuis la spécification jusqu'à la génération du code correspondant à l'implantation finale, nous avons donc eu besoin d'enrichir le modèle d'algorithme présenté jusqu'ici afin de permettre la construction du graphe de voisinage (Cf. chapitre 4), graphe intermédiaire nécessaire à la synthèse du contrôle.

Nous avons vu que chaque sommet correspond à une opération de calcul, d'entrée-sortie ou de factorisation et que les dépendances de données correspondent aux données transmises entre les opérations. Plusieurs dépendances de données peuvent aboutir à un même sommet (consommateur) ou être issues d'un même sommet (producteur). Comme ces sommets peuvent être situés soit du côté lent $'s'(slow)$ de la frontière, soit de son côté rapide $'f'(fast)$, l'ensemble des dépendances de données les reliant peuvent donc provenir d'un sommet producteur du côté lent 's' (ou rapide 'f') vers un ou plusieurs sommets consommateurs du côté lent 's' (ou rapide 'f').

En effet, ces dépendances de production et consommation de données au niveau des frontières entre leurs côtés lents et rapides conditionnent les relations entre frontières (appelées relations de voisinage) qui sont nécessaires à la génération du contrôle.

Il faut donc être capable de générer les signaux de contrôle dans l'ordre qui correspond à celui des dépendances de relations de voisinage entre frontières en précisant les côtés (lent/rapide) des provenances et des destinations.

Tel que formalisé précédemment, le modèle de graphe d'algorithme ne permet pas de stocker l'information concernant les côtés de provenances et de destinations des dépendances de données. De plus, lors de la génération du graphe de voisinage, mais aussi pour la phase d'établissement du contrôle lors de l'implantation, il est indispensable de connaître le type d'une dépendance, c'est à dire son côté de provenance et son côté de destination.

Pour générer automatiquement ce graphe de voisinage et le chemin de contrôle du circuit, nous proposons de valuer chaque dépendance de donnée par un couple modélisant le type de côté de son producteur et de son consommateur.

4.4 Conclusion

Dans ce chapitre, nous avons commencé par un état de l'art sur la spécification et la modélisation des systèmes embarqués temps réel, dans le but de situer notre modèle de spécification par rapport aux modèles existants. Nous avons ensuite défini et décrit succinctement notre modèle de graphes factorisés et conditionné de dépendances de données qui constituera le point de départ du processus d'implantation matérielle de l'application sur l'architecture cible.

L'intérêt de ce modèle basé principalement sur le modèle flot de données réside dans sa capacité à maintenir l'importance des données dans le traitement tout en offrant plus de flexibilité dans les spécifications des structures de contrôle pour les itérations (For Loop), et l'exécution conditionnelle (IF, CASE).

Ce graphe peut être spécifié directement comme décrit ci-dessus à l'aide d'une interface graphique ou textuelle comme celle du logiciel SynDEx ou produit par les compilateurs des langages synchrones Esterel, Lustre, Signal, à travers leur format commun, DC, qui est aussi un graphe flot de données. Cette dernière approche présente l'intérêt de permettre d'effectuer des vérifications formelles sur l'algorithme en se servant des mécanismes de validation associés à ces langages. Ces vérifications, effectuées très tôt dans le cycle de développement, permettent d'éliminer un grand nombre d'erreurs concernant la logique des algorithmes de l'application qui sont très difficiles à corriger au niveau de leur implantation sur l'architecture cible.

Chapitre 5

Modèle d'implantation

Ce chapitre concerne la deuxième étape du flot d'extension d'AAA que nous proposons dont l'objectif est de spécifier l'approche de génération de la solution d'implantation matérielle. Il s'agit de présenter notre modèle d'implantation basé sur un ensemble de transformations de graphes permettant la synthèse du support matériel d'exécution selon un modèle RTL et des mécanismes de transferts de données synchronisés. Nous présentons succinctement l'approche particulière que nous avons préconisée pour résoudre les deux problèmes sous-jacents à l'activité de synthèse : la synthèse du chemin de données et celle du chemin de contrôle.

5.1 Introduction

Le processus d'implantation matérielle consiste en la transformation des spécifications comportementales de l'application sous forme algorithmique en une architecture matérielle capable de les exécuter. Comme les outils classiques de CAO permettent un passage quasi automatique d'un modèle RTL de l'architecture (niveau transfert de registres *"Register Transfer Level : RTL"*) à la puce, la grande difficulté qui se pose actuellement lors de l'implantation est d'assurer le passage aisé du modèle comportemental ou algorithmique de l'application au modèle RTL de l'architecture et cela sous de fortes contraintes de qualité et de temps de conception. Réaliser une implantation matérielle revient donc à définir un flot systématique et automatisé d'implantation allant de la spécification de l'application à la génération de l'architecture RTL du circuit tout en maîtrisant la complexité de ce flot ainsi que la complexité de l'architecture RTL générée.

Cette étape de conception consistant à générer l'architecture au niveau RTL à partir d'une spécification au niveau comportemental ou algorithmique est communément appelée *synthèse comportementale* (aussi appelée *synthèse de haut niveau* ou *synthèse d'architecture*). Cette synthèse définie une méthodologie de conception qui détermine la séquence des étapes ainsi que les informations transmises entre ces étapes pour accomplir la génération de l'architecture circuit réalisant l'implantation matérielle. Il est clair qu'un tel processus permet d'automatiser la recherche de l'architecture satisfaisant au mieux l'ensemble des contraintes imposées (telles que le temps de propagation, la surface ou la consommation) et de réduire considérablement la durée du cycle de développement des circuits numériques.

Plus particulièrement, les outils de synthèse d'architecture permettent l'implantation matérielle sous contraintes des algorithmes des applications et fournissent rapidement des estimations de surfaces et des performances. Ils définissent ainsi un procédé automatisé qui facilite l'adéquation algorithme-architecture en permettant d'une part d'explorer l'espace des solutions architecturales à partir d'une description comportementale de l'algorithme en vue de la recherche d'une solution architecturale "optimale" vis à vis des contraintes spécifiées et d'autre part de favoriser une appréhension immédiate des conséquences architecturales de toute modification de l'algorithme. Compte tenu du niveau d'abstraction élevé des spécifications (niveau comportemental), ces outils de synthèse d'architecture permettent d'envisager de nouvelles perspectives à l'adéquation entre algorithme et architecture.

Toutefois, l'évolution conjointe des technologies ainsi que l'augmentation de la complexité des applications implique une modification des modèles et techniques définis dans les processus de synthèse architecturale et l'adaptation de nouvelles techniques basées sur des modèles formels en vue de fiabiliser le cycle de développement, et de garantir une parfaite maîtrise, dès le niveau algorithmique, des performances et des coûts de l'architecture.

Pour ce faire, nous proposons dans le cadre de notre travail d'extension d'AAA, une nouvelle méthode qui repose sur un ensemble de règles et de transformations formelles basées sur la théorie des graphes et des ordres partiels durant tout le processus de synthèse d'architecture.

Dans la suite de ce chapitre, nous présentons cette approche de synthèse, en particulier, ses règles qui permettent la synthèse du chemins de données et de contrôle du circuit RTL correspondant à l'algorithme spécifié sous forme de graphe factorisé et conditionné de dépendances des données. Jusqu'au début des années 90, on croyait que la synthèse du chemin de contrôle était la plus difficile à réaliser [**?**]. Plusieurs travaux ont depuis démystifié cette difficulté [98, 99]. Nous montrons ici qu'il est possible de synthétiser le chemin de contrôle, de manière simple et systématique, à l'aide d'une technique de synchronisation

des transferts de données entre registres. Cette approche nous a permis par la suite de réaliser facilement un générateur automatique de VHDL structurel synthétisable, réalisant ainsi la synthèse automatique de circuits.

Nous y présentons également un état de l'art sur la synthèse de circuits et, plus particulièrement, sur la synthèse aux niveaux d'abstraction *comportemental* et *transfert de registres*, qui sont les niveaux ciblés par ce travail. Afin de pouvoir situer les apports et limites de notre approche par rapport aux outils et méthodes existants, nous décrivons succinctement quelques outils de synthèse.

5.2 Synthèse de circuits : présentation générale des concepts

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

Durant le processus de conception descendant d'un circuit, on distingue généralement différents niveaux de description, plus ou moins abstraits, par rapport aux détails du circuit (voir figure 5.1). Ces niveaux d'abstraction définissent différents niveaux de synthèse classés selon le niveau d'abstraction qu'ils manipulent et selon les domaines de description qu'ils couvrent.

Nous décrivons ci-dessous les différents niveaux de synthèse et les principaux outils de synthèse correspondant à chacun de ces niveaux.

5.2.1 Synthèse au niveau système

Le niveau système correspond au niveau d'abstraction le plus haut. Il permet de spécifier un système entier en termes de sous-systèmes. Chaque sous-système est considéré comme un processus communicant (CSP), synchronisé par l'intermédiaire des échanges de messages entre eux. Après le partitionnement, chaque processus peut être représenté au niveau comportemental par un graphe de flot de données et de contrôle (CDFG Control Data Flow Graph) synchronisé par les événements d'entrée/sortie. La synthèse système permet donc de déterminer la description comportementale et de partitionner le système à synthétiser en sous-circuits.

Plusieurs environnements de conception au niveau système ont été développés récemment. Citons en particulier l'environnement *Tosca* [101, 100] qui cible les systèmes orientés-contrôle, les environnements *SpecSyn* [102], *Vulcan II* [104] et Cosyma [69, 105]. Pourtant, il y a encore peu d'outils commerciaux de synthèse *niveau système* disponibles, puisqu'ils sont encore au stade de la recherche.

5.2.2 Synthèse au niveau comportemental

Ce niveau appelé aussi niveau algorithmique vise à décrire seulement la fonctionnalité d'un circuit à l'aide de langages séquentiels ou procéduraux. En général, c'est une description algorithmique du comportement du circuit qui ne contient aucune information ni sur sa structure ni sur la façon dont il sera réalisé.

La synthèse comportementale part donc d'une spécification algorithmique en entrée, décrite par un langage procédural (VHDL, Verilog) ou applicatif (*Silage* [66]), ou par l'intermédiaire des modèles CFG (*Control Flow Graph*), DFG (*Data Flow Graph*), CDFG ou FSMD (*FSM with Datapath*). Cette

spécification est traduite de façon automatique ou semi-automatique (traduction structurelle) dans une description matérielle au niveau transfert de registre (RTL pour Register Transfer Level), représentée fréquemment sous la forme d'une architecture contrôleur/chemin de données.

Ce processus de traduction détermine l'assignation des fonctions du circuit aux opérateurs appelés aussi ressources, les interconnexions entre les différents opérateurs, ainsi que le moment d'exécution de chaque opération dans des intervalles de temps appelés "pas de contrôle". La synthèse peut s'effectuer en considérant soit que la surface est bornée soit en considérant que la latence est limitée.

Notons en particulier que plusieurs travaux de recherche universitaires concernant les outils de synthèse comportementale ont été objets de transferts de technologie à l'industrie, comme *Amical* [106, 107] et *FPGA Express* [108] de *Viewlogic*, mais le degré d'acceptation de ces outils en milieu industriel n'a jamais réussi à atteindre celui de la synthèse au niveau transfert de registre, il reste encore beaucoup de progrès à faire pour surpasser les problèmes posés par les outils classiques et atteindre le degré d'acceptation escompté. Nous présentons, dans la section 5.3.2, les principaux outils de synthèse *comportementale* existant actuellement.

5.2.3 Synthèse au niveau transfert de registres

À ce niveau, les opérations sont interprétées comme des transferts de données entre registres traversant des circuits transitoires. Les objets manipulés sont ainsi liés directement à des réalisations physiques. L'objectif de la synthèse *transfert de registres* est de traduire une description de niveau RTL en une description (de la réalisation) au niveau portes logiques définies par la technologie. Le format intermédiaire utilisé à ce niveau de synthèse est généralement sous la forme de FSM (*Finite State Machine*), de BDD (*Binary Decision Diagram*) ou d'équations booléennes, contenant une partie contrôle et une partie chemin de données.

Le chemin de données est composé de trois types de composants : les unités fonctionnelles (ULA– Unité Logique-Arithmétique, multiplieurs et registres à décalage), les unités de stockage (registres et mémoires) et les unités d'interconnexion (bus et multiplexeurs). Le contrôleur spécifie l'ensemble de micro-opérations appliquées sur le chemin de données pendant chaque étape de contrôle. Au niveau RTL, les transferts de données sont synchronisés par les fronts du signal d'horloge. La synthèse RTL met en correspondance les composants du contrôleur et du chemin de données avec une bibliothèque de cellules, afin de produire une *netlist* de portes. Elle détermine de ce fait la structure microscopique du circuit, c'est-à-dire l'interconnexion de portes.

Plusieurs outils de synthèse RTL sont disponibles, tels que *Synopsis* [112] , *Compass* [113], *Viewlogic* [108], etc.

5.2.4 Synthèse au niveau portes ou synthèse logique

Elle détermine la vue structurelle d'un circuit au niveau logique. À ce niveau, le circuit est décrit comme un ensemble de bascules et de portes logiques interconnectées. Les outils de synthèse *logique* partent d'une description structurelle sous la forme de blocs logiques (combinatoires et registres) interconnectés et synchronisés par un signal d'horloge. Leur but est d'optimiser ces blocs logiques en termes de surface du circuit généré. Les outils de synthèse logique font appel à des bibliothèques de composants technologiques qui seront utilisés pour l'implantation. La dernière étape de la synthèse qui détermine quelles portes de la bibliothèque seront utilisées est appelée "mapping technologique".

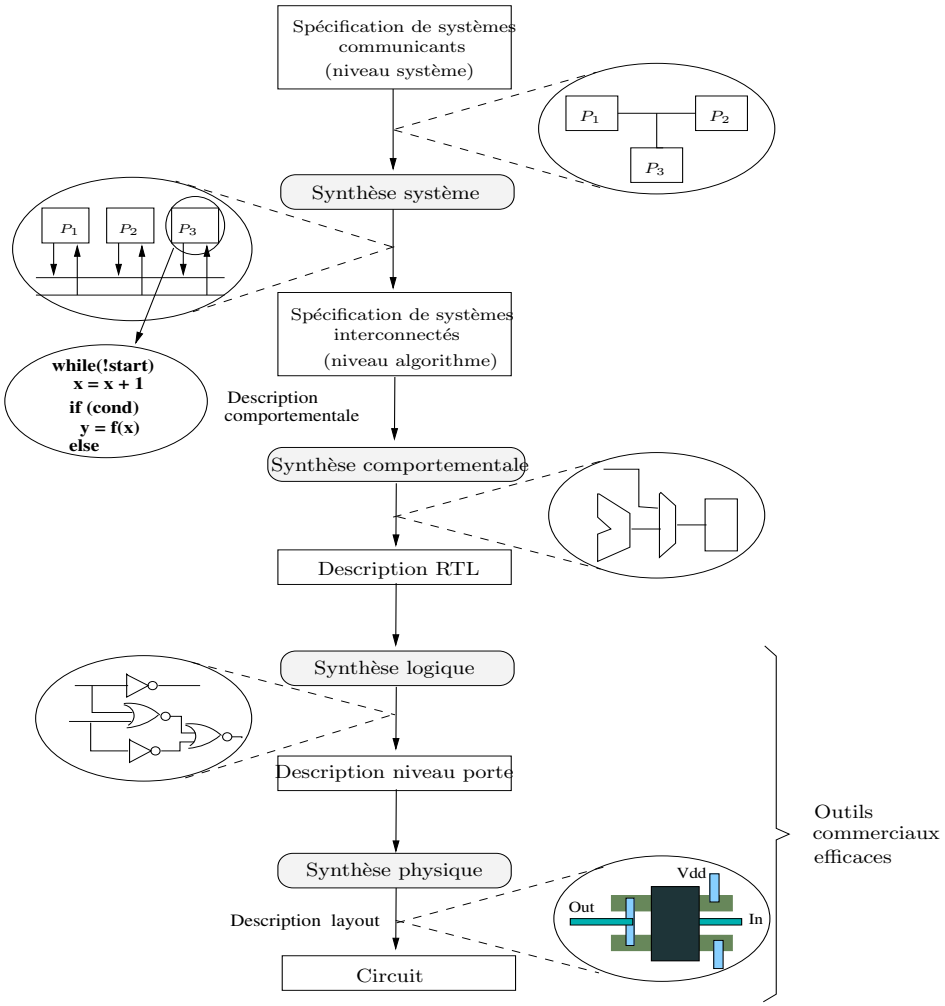


FIG. 5.1 – Les niveaux de synthèse

5.2.5 Synthèse au niveau transistor ou synthèse physique

Ce niveau correspond au niveau d'abstraction le plus bas de la conception de circuits intégrés, il se présente comme l'interconnexion de composants électroniques, tels que les transistors, les diodes.

La synthèse *physique* a pour rôle d'associer à une description dans le domaine structurel, sous la forme d'une *netlist* de portes ou de modèles de *layout*, une représentation dans le domaine physique afin de produire le *layout* final de l'implantation. À ce niveau-là la synchronisation est effectuée par les changements des valeurs des conducteurs. La description du layout est la dernière étape de conception d'un circuit intégré, le fondeur utilisera ce layout lors de la fabrication du circuit intégré.

Comme nous le verrons par la suite, notre méthodologie se situe entre les niveaux comportemental et RTL, puisque la synthèse du chemin de données est effectuée à partir d'un modèle basé sur des

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

5.3 Synthèse comportementale

La synthèse comportementale est une méthodologie de conception qui accepte comme modèle d'entrée une spécification comportementale ou fonctionnelle et produit une architecture capable d'exécuter cette spécification initiale. Ce procédé de synthèse automatise et optimise la transformation d'une description comportementale en une description RTL. L'architecture résultante généralement décrite au niveau transfert de registres est composée d'un contrôleur et d'un chemin de données.

Le rôle de ce procédé automatisé de synthèse est de trouver le nombre et le type de ressources matérielles nécessaires à l'implantation des opérations (allocation), de définir les instants d'exécution des opérations dans des intervalles de temps appelés "pas de contrôle" et de définir pour chaque opération quelle est la ressource qui l'implante (assignation). En d'autres termes, la synthèse comportementale définit un modèle structurel du chemin de données comme une interconnexion de ressources et un modèle de niveau logique de l'unité de contrôle. Cette unité de contrôle est chargée de piloter les signaux du chemin de données.

Les principaux avantages de la synthèse au niveau comportemental sont les suivants :

- une réalisation rapide de la spécification, car la description comportementale est plus courte (environ 10 fois plus courte qu'une description RTL) et plus intuitive (instructions similaires à l'algorithme de départ) ce qui diminue aussi le nombre d'erreurs et rend la maintenance de code plus facile,
- une simulation et par conséquent une vérification fonctionnelle plus rapide, car la description est plus abstraite qu'une description RTL,
- la possibilité de créer et d'évaluer plusieurs implémentations rapidement et donc de déterminer le meilleur compromis entre la latence et la surface du circuit,
- une meilleure qualité de résultats, car durant la synthèse, l'outil peut optimiser le compromis entre la surface et la latence en consultant la surface et les délais estimés de chaque composant qu'il utilise,
- réduction considérable du délai de mise sur le marché (*time to market*) étant donné la possibilité pour des non-experts de concevoir des circuits pour des applications particulières.

La synthèse comportementale, aussi appelée synthèse d'architecture, permet donc l'implantation matérielle sous contraintes des algorithmes et fournit une méthodologie de synthèse très efficace en termes de gestion de la complexité des algorithmes de l'application et en termes de temps de conception par rapport à une synthèse RTL. Cette possibilité de créer et produire rapidement plusieurs implantations par rapport à une synthèse RTL permet une exploration efficace de l'espace des solutions architecturales à partir d'une description comportementale de l'algorithme et facilite de ce fait la synthèse à partir d'un très haut niveau d'abstraction.

En effet, dans la spécification comportementale, il n'est plus nécessaire de décrire les registres de

mémoires, les opérateurs tels que les additionneurs et les multiplieurs, les multiplexeurs, et même le contrôleur qui supervise les opérations, la mémoire, et les entrées/sorties. Le concepteur algorithmicien n'a plus besoin de spécifier l'architecture exacte du circuit et peut automatiquement explorer plusieurs implantations pour trouver une architecture optimale vis à vis des contraintes spécifiées.

Cette capacité à spécifier et valider des systèmes à un haut niveau d'abstraction (niveau comportemental) permettra aux concepteurs d'algorithmes de prototyper (sous forme virtuelle) leurs applications et aux concepteurs de systèmes de concevoir rapidement ceux-ci, tout en explorant un large spectre de solutions, tant architecturales que technologiques.

D'autre part, l'approche consiste à intégrer au plus haut niveau le savoir-faire des spécialistes en conception de circuits VLSI, à travers l'utilisation de composants spécifiés à un niveau comportemental.

Dès lors, grâce au haut niveau d'abstraction des spécifications, la synthèse comportementale permet d'envisager de nouvelles perspectives : d'une part l'algorithmicien fournit un ensemble de spécifications génériques, flexibles et réutilisables, d'autre part le concepteur de systèmes VLSI intègre cet algorithme dans un environnement applicatif en précisant les différents paramètres.

5.3.1 Stratégies de synthèse comportementale

Comme nous venons de le présenter ci-dessus la synthèse comportementale, autrement dit la synthèse de haut niveau correspond à un ensemble de tâches de raffinements ou de transformations qui à partir d'une spécification comportementale, génèrent une architecture cible capable d'exécuter la spécification algorithmique initiale. L'architecture résultante composée d'un chemin de contrôle et d'un chemin de données est souvent décrite au niveau transfert de registres. La description comportementale initiale définit la fonctionnalité qui doit être réalisée par le système synthétisé. Cette description peut avoir une représentation textuelle ou graphique. Elle peut être générée automatiquement par un outil de synthèse de plus haut niveau ou décrite par le concepteur à partir d'une spécification. Elle est d'abord compilée en une représentation interne intermédiaire qui dépend des algorithmes de synthèse utilisés dans l'outil. Les étapes de synthèse s'appliquent ensuite à cette représentation pour transformer la structure initiale graduellement vers une architecture cible. En général, ces étapes sont pilotées par des contraintes d'utilisation qui influencent certaines caractéristiques des algorithmes ainsi que l'architecture résultante.

Dans un flux de synthèse, ces étapes peuvent se faire en suivant deux stratégies différentes :

(i) la synthèse automatique ou synthèse par allocation-ordonnancement : dans cette approche classique, le flux de synthèse effectue la compilation de la description comportementale en une représentation interne souvent sous la forme d'un graphe de flot de données et de contrôle (CDFG), puis applique des transformations de haut niveau, telles que l'ordonnancement, l'allocation des opérateurs (ressources), l'association (affectation) des opérateurs aux opérations et finalement, la génération d'une architecture composée d'un chemin de contrôle et d'un chemin de données. Cette approche traditionnelle connue depuis le début les années 80 soulève plusieurs problèmes qui ont empêché que la synthèse comportementale soit répandue et acceptée pour les applications industrielles.

(ii) la synthèse par transformations successives : cette approche considère la synthèse comme une problématique de transformation de code et cherche à appliquer une approche basée sur la transformation de programmes à la synthèse d'architectures matérielles. Les transformations successives consistent à modifier progressivement les spécifications comportementales, par réécritures successives et prouvées,

jusqu'à obtenir une implantation matérielle qui respecte les contraintes de l'application. Le but de cette technique de synthèse par transformations successives est d'améliorer l'efficacité de la synthèse, tout en préservant la sémantique et la correction de la spécification initiale. Il existe des outils généralistes comme *Transe* [115, 114] et des outils de mise en œuvre des algorithmes sur des architectures régulières comme *Alpha du Centaur* [117, 116, 118], *Presage* [120], etc.

Le grand avantage de l'environnement expérimental *Transe* qui adopte cette approche est qu'il permet d'obtenir, par dérivations et à partir d'une spécification en langage *Lustre*, la description de son implantation matérielle en même temps que la preuve formelle de sa conformité [114]. Les programmes Lustre sont, de façon semi-automatique, transformés progressivement et interactivement par réécritures successives, en utilisant des schémas de transformation prouvés à l'avance. L'objectif est d'obtenir un programme interprétable structurellement en termes d'architecture matérielle et possédant de bonnes propriétés opérationnelles.

Les transformations qu'elles soient automatiques de type allocation-ordonnancement ou semi-automatique par transformations successives prouvées cherchent à optimiser la spécification initiale et à rendre le comportement du circuit le plus proche possible de l'optimum. Ces transformations peuvent s'appliquer sur le flot de contrôle ou sur le flot de données. Quelques transformations se sont inspirées de celles réalisées par les compilateurs (l'élimination de code inutile, la propagation de constantes, l'élimination de sous-expressions communes, le déroulage de boucles, l'expansion de corps de procédures, etc.). D'autres ne concernent que la synthèse comportementale, comme la transformation d'une multiplication par puissance de deux dans un décalage, la réduction du nombre de niveaux dans les graphes de contrôle et de données, l'accroissement du parallélisme, etc.

Dans la méthodologie que nous proposons, nous effectuons la synthèse comportementale par l'intermédiaire de transformations spatio-temporelles (défactorisations : comme nous le verrons dans le chapitre 5). Ces transformations, réalisées à l'aide d'une heuristique d'optimisation, préservent la sémantique de la spécification initiale, puisqu'elles sont basées sur l'équivalence entre les opérateurs de factorisation de motifs répétitifs (*FORK*, *JOIN* et *ITERATE*) et leurs correspondants défactorisés (*EXPLODE* et *IMPLODE*). Cela nous permettra de trouver l'architecture qui respecte les contraintes temporelles et qui minimise l'augmentation du nombre de ressources matérielles utilisées. À la fin de cette étape de synthèse, nous avons un graphe matériel, obtenu par traduction directe du graphe algorithmique transformé en faisant correspondre à chaque sommet du graphe d'algorithme un opérateur et à chaque arc une connexion entre opérateurs, et un ensemble d'équations, obtenu par analyse des relations de voisinage entre les frontières de factorisation du graphe algorithmique. Cet ensemble d'équations logiques permet de synthétiser la partie contrôle de l'architecture. La synchronisation entre les sommets du graphe matériel est assurée par un signal d'horloge. Nous effectuons donc une synthèse automatique par transformations successives (basées sur la théorie des graphes) en mélangeant les deux stratégies présentées ci-dessus.

Afin de permettre de situer notre approche par rapport aux outils et méthodes existantes nous allons décrire par la suite quelques systèmes et outils de synthèse.

5.3.2 Outils de synthèse comportementale

Au cours des dernières années une très forte activité de recherche dans le domaine de la synthèse comportementale (ou synthèse de haut niveau, ou synthèse architecturale) a donné lieu à plusieurs outils de synthèse. Cependant ce sont encore, pour la plupart, que des travaux universitaires en cours de développement (Cathedral IMEC-Belgique, Hyper UCB-Berkeley Californie, GAUT ENSSAT-France,...). Parmi eux, très peu offrent un outil complet, et encore moins sont parvenus à atteindre le niveau de commercialisation : Behavioral Compiler de Synopsys, Monet de Mentor Graphics Corporation, Visual Architect de Cadence etc... Cependant, si leur utilisation dans le secteur industriel est encore plutôt au stade de la prospective, ces outils permettront sans doute d'améliorer le gain de productivité. Il reste cependant encore beaucoup d'efforts à faire au niveau recherche, notamment dans l'apport de prise en compte de nouvelles contraintes, l'amélioration des résultats de la synthèse et l'automatisation du processus de synthèse...

Pour présenter les principales caractéristiques des outils existants, nous avons choisi quelque-uns des outils de synthèse. Ces outils ne sont pour la plupart qu'au stade de développement universitaire, il nous a paru nécessaire d'étudier principalement les outils universitaires Gaut et Cathedral (outils orientés principalement flot de données), Amical et Hyper (outils orientés flot de contrôle ou mixte). Ce qui nous a permis d'en retirer leurs principales caractéristiques que nous résumons dans le tableau récapitulatif final Tab.5.3.2).

GAUT

GAUT est un outil développé conjointement au Laboratoire d'Analyse des systèmes de Traitement de l'Information (LASTI) à Lannion et au Laboratoire d'Etude des Systèmes Temps Réel (LESTER). Cet outil est dédié aux applications de traitement du signal et de l'image sous contrainte de temps d'exécution. Il cible principalement la conception sur FPGA (Field Programmable Gate Array) mais peut aussi produire des ASICs (Application Specific Integrated Circuit).

Le concepteur fournit à l'outil GAUT une description comportementale en VHDL du circuit, une bibliothèque de modèles caractérisés, une cadence ou une fréquence de fonctionnement : les deux derniers paramètres constituent le temps d'exécution total visé par le concepteur. La cadence des calculs, pouvant être liée à une fréquence d'échantillonnage, constitue la contrainte temporelle principale qu'il faut respecter par l'algorithme de synthèse. Une fois cette contrainte respectée, la surface et la consommation doivent être minimisées.

Pour évaluer le nombre de cycles total de la description comportementale, GAUT déroule toutes les boucles. Les branchements conditionnels sont cablés, le chemin d'exécution est alors le plus long des deux branchements. En effet, GAUT est le seul outil qui utilise un graphe uniquement flot de données, et n'intègre donc pas d'unité de contrôle permettant de prendre les conditions. Il est donc obligé d'intégrer le contrôle dû à un traitement conditionnel dans son chemin de données par surdimensionnement du graphe flot de données en prévoyant toutes les possibilités des corps de programme. Les boucles non bornées ne sont pas acceptées par GAUT, ce qui limite le champs d'application de l'outil.

Après une phase de compilation permettant l'extraction du parallélisme, le graphe flot de données obtenu sera matérialisé selon un modèle de coeur de processeur de traitement du signal constitué de 4 unités fonctionnelles : l'unité de traitement, l'unité de contrôle, l'unité de mémorisation et l'unité de communication. Enfin la synthèse aboutit à la génération d'une description VHDL structurelle de

l'architecture conçue. Ce VHDL de sortie est interprétable directement par les outils de synthèse logique COMPASS ou SYNOPSYS.

Cathedral 2/3

Cathedral 2/3 est un outil développé par les laboratoires de l'IMEC et de l'ESAT situés en Belgique. Il s'agit d'un outil de synthèse spécifique aux algorithmes de traitement du signal ou DSP (Digital Signal Processing) en temps réel pour les réaliser en ASICs. Le concepteur fournit une description en Silage [66] d'un circuit de type flot de données et décide de la fréquence de son circuit. Cette description en Silage est traduite en C à l'entrée de Cathedral 2/3 pour sa vérification fonctionnelle. La synthèse par contre est effectuée à partir du graphe flot de données issu du programme Silage, et non du C qui a servi à la validation. L'outil est constitué de deux parties : Cathedral 2 et Cathedral 3. Cathedral 2 est l'ossature du programme et Cathedral 3 est un générateur d'opérateurs. Ces opérateurs sont de deux types : les EXU (Execution Unit) et les ASU (Application Specific Unit).

- Les EXU sont issues de la bibliothèque de l'outil. Il peut y avoir au maximum quatre EXU instanciées dans le circuit. Une EXU peut être soit une ALU, soit une ROM.
- Les ASU sont crées par Cathedral 3 sur demande, principalement pour accélérer l'exécution. Ces ASU sont utilisées comme opérateurs par Cathedral 2 au même titre que les EXU. Elles sont instanciées pour des opérations complexes qui se répètent un certain nombre de fois (le flot d'opérations fonctionnelles d'un corps de boucle par exemple). Chaque ASU possède un contrôleur local pour éviter la perte d'un cycle dans le contrôleur global.

Après synthèse, Cathedral 2/3 génère un circuit constitué d'un chemin de données et d'un contrôleur micro-programmé. Il peut être comparé à un processeur VLIW avec contrôleur micro-programmé. Ce circuit est décrit en VHDL au niveau RTL.

Il est bon de noter que la contrainte temps réel lui interdit comme Gaut d'utiliser des boucles non bornées. De plus il manipule encore très mal les tableaux et boucles. Il est donc nécessaire de dérouler toutes les boucles "manuellement" dans la description comportemenatle de l'algorithme, ce qui peut se révéler rapidement très lourd.

Amical

Amical est un outil développé au laboratoire Techniques de l'Informatique et de la Micro-électronique pour l'Architecture d'ordinateurs (TIMA) de Grenoble. Cet outil s'adresse aux applications dominées par les flots de contrôle pour les réaliser sur des ASICs ou des FPGAs. En partant d'une description comportementale en VHDL fournie en entrée et d'une bibliothèque de modèles, Amical produit une description destinée aux outils de synthèse logique existants. La description comportementale d'entrée doit être constituée d'un unique processus VHDL qui contient des instructions séquentielles ainsi que des appels procéduraux. Le VHDL accepté en entrée est assez complet (wait, loop, exit, pas de For,...) permettant une véritable description comportementale. Les fonctions utilisées dans la description sont des boîtes noires faisant référence aux modèles de la bibliothèque. Les informations sur ces boîtes noires sont, par exemple, les opérations qu'elles sont capables d'exécuter et le protocole qu'elles suivent. Cette approche laisse une grande flexibilité à la réutilisation et l'étend même au niveau comportemental : la bibliothèque utilisée par Amical peut inclure des modèles avec des méga-fonctions telles que des opérateurs DSP, des coeurs CPU, des convertisseurs AD/DA,... La description comportementale a des

liens avec ces blocs de méga-fonctions grâce aux fonctions et procédures appelées dans le programme.

Après une synthèse basée sur l'ordonnancement et l'allocation, Amical, génère un circuit constitué d'un contrôleur central synchrone (automate d'états) et d'un chemin de données avec des bus pouvant inclure plusieurs unités fonctionnelles travaillant aux même temps.

Notons enfin que l'outil permet d'alterner la conception manuelle et automatisée. Le concepteur peut contrôler chaque étape de la synthèse et a plus de liberté à explorer les solutions architecturales pour déterminer son architecture.

Hyper

L'outil Hyper, développé à l'université de Berkeley (USA), est principalement dédié aux applications temps réel. À partir d'une description faite en Silage, de contraintes de temps, ainsi que d'une bibliothèque de composants, Hyper cherche à trouver l'implantation matérielle ayant une surface ou un temps de latence minimal. Pour ce faire la description d'entrée est analysée puis compilée en un graphe flot de données et de contrôle (CDFG) intermédiaire. Ce CDFG représente l'algorithme comme un graphe flot de données étendu avec un flot de contrôle qui matérialise les états tels que les boucles ou bien les structures conditionnelles. Il est utilisé comme base de données centrale sur laquelle reposent toutes les opérations de synthèse telles que les estimations, les transformations (*retiming, software pipelining...*), l'allocation ou l'ordonnancement.

Les boucles non bornées ne sont pas acceptées par Hyper et il manipule encore très mal les tableaux et boucles. En effet ces dernières sont déroulées complètement lors de la description comportementale, ce qui peut se révéler rapidement très lourd.

La synthèse aboutit à la génération d'une description SDL ou VHDL structurelle de l'architecture. Le SDL ainsi synthétisé peut alors être interprété directement par le compilateur de silicium *LARGER IV* qui est également un outil universitaire issu de Berkeley. La sortie VHDL est quant à elle destinée à l'outil de synthèse logique de *SYNOPSYS*. L'architecture ciblée par Hyper est composée d'un rassemblement des différents chemins de données, d'une machine à états finis faisant office de contrôle central, et d'une interface logique s'occupant du contrôle local.

À travers l'étude de ces outils de synthèse nous avons cherché à couvrir un large spectre des travaux effectués dans le domaine de la synthèse comportementale en sélectionnant les outils représentatifs des différentes classes d'outils de synthèse : ceux qui suivent une approche orientée flots de données et ceux qui suivent une approche orientée flot de contrôle ou mixte. L'objectif n'est évidement pas de dire si tel ou tel outil est meilleur qu'un autre, mais de créer un état de l'art des différents procédés de synthèse utilisés dans ces outils et de déterminer leurs principal avantages et limites. Notons en particulier que tous ces outils possèdent des particularités qui peuvent se révéler bénéfiques aux types d'algorithmes, aux domaines d'applications auxquels ils sont dédiés, ou désavantageux pour les autres domaines. Cette diversité d'objectifs rend très difficile toute tâche d'évaluation comparative indépendante du domaine d'application.

Nous pouvons tout de même faire certaines constations : la contrainte temps réel interdit à tous les outils d'utiliser des boucles non bornnées. À priori il y a deux manières de traiter les boucles finies. Soit on les déroule intégralement lors de la compilation, soit on synthétise le coeur et on crée un compteur de

TAB. 5.1 – Caractéristiques des outils de synthèse comportementale

Outil	*Gaut*	*Cathedral*	*Amical*	*Hyper*
Langage d'entrée	VHDL : sous-ensemble réduit	Silage : sous-ensemble	VHDL assez complet	Silage : sous-ensemble
Représentation interne	Graphe flot de données uniquement	Graphe flot de données	Graphe flot de contrôle	Graphe flot de données et de contrôle
Contraintes de synthèse acceptables	Temps de latence qui correspond à la cadence des calculs. La surface et la consommation en second lieu	Temps de latence puis surface	Surface puis consommation	Temps de latence ou surface : au choix de l'utilisateur
Traitement des conditions et boucles	Déroulement manuel complet de toutes les boucles. Traitement non encore efficace du conditionnement. L'outil ne connaît que le type integer pas de booléens.	Déroulement manuel de toutes les boucles. Traitement des instructions conditionnelles 'if' ou 'while'	Le VHDL utilisable en entrée n'accepte pas les 'For'	Synthétise le coeur des boucles et crée un compteur de boucle : pas de déroulement, et traitement des instructions conditionnelles 'if' ou 'while'.
Modèle d'architecture	Modèle générique de coeur de processeur de traitement du signal à base d'unités fonctionnelles communiquant entre elles par des multi-bus	Processeur VLIW avec contrôleur micro-programmé	Modèle de machine d'états finis avec des co-processeurs, interconnectés par des Bus.	Contrôleur réalisé par une machine à états finis (FSM) et un rassemblement de chemins de données incluant des unités d'éxecution (EXUs), des registres ou Mux...
Langage en sortie	VHDL niveau RTL compatible COMPASS et SYNOPSYS	Langage interne interprétable par le compilateur de CADENCE. VHDL : sortie optionnelle	VHDL niveau RTL (Synopsys, Synergy,..).	VHDL niveau RTL (Synopsys) ou SDL (LARGER IV)
Technologie cible	FPGAs, ASICs	ASICs	FPGAs, ASICs.	ASICs
Notes	Architecture adaptée au traitement du signal	Seul outil à décrire son architecture au niveau porte	Outil alternant conception manuelle et automatisée, bonne estimation de la consommation	Seul outil à permettre la simulation directe depuis son interface. Estimation des caractéristiques énergétiques

boucle. Gaut et Catheral déroulent toutes les boucles tandis que Hyper synthétise leur coeur et ne les déroule donc pas. D'autres, comme Amical ne permettent même pas la manipulation de tableaux à plus d'une dimension (limité aux vecteurs). Il est à remarquer aussi que même si "théoriquement" le Silage accepté par Hyper et Cathedral permet l'utilisation de boucles, ces outils ne les gèrent pas efficacement. En effet, ces derniers manipulent encore très mal les tableaux et boucles. Il est donc nécessaire de dérouler toutes les boucles "manuellement" dans la description comportementale de l'algorithme. Ce défaut peut se révéler très gênant lors de la description d'algorithmes nécessitant des opérations très répétitives ce qui est le cas de nos applications cibles en traitement du signal et d'images. On note également que tous ces outils sont capables d'effectuer du traitement conditionnel en utilisant les instructions If ou While. Seul Gaut est obligé d'intégrer le contrôle dû au traitement conditionnel dans son chemin de données étant donnée l'approche purement flot de données qu'il adopte. Du point de vue du contrôle généré, tous les outils suivent une stratégie "centralisée" à une exception près pour Cathedral et Hyper qui tentent tout de même d'allier contrôle centralisé et décentralisé en utilisant un contrôleur central global et un ensemble de contrôleurs locaux. Cependant, il est généralement admis qu'une logique de contrôle "centralisée" est souvent plus complexe à maîtriser en particulier le problème d'explosion d'états des machines à états finis (FSM).

Bien que nous prétendions pas proposer un outil de synthèse complet et meilleur que les autres, ce qui n'est bien évidement pas l'objectif de notre travail, cependant l'approche de synthèse que nous présentons dans le cadre de notre travail d'extension d'AAA présente des caractéristiques qui répondent à la majorité des lacunes citées précédemment. En fait, l'approche de synthèse proposée, qui fera l'objet d'une présentation détaillée par la suite, supporte bien le traitement conditionnel ainsi les traitements répétitifs au sein des boucles sans recours à un déroulage préalable. De plus la logique de contrôle générée est délocalisée et consiste à associer à chaque frontière de factorisation sa propre unité de contrôle. Ce qui permet aux outils de CAO utilisés pour la synthèse de placer les unités de contrôle plus proches des opérateurs à contrôler.

5.4 Synthèse AAA

Dans ce travail d'extension de AAA que nous proposons, le processus de synthèse part d'une spécification algorithmique sous la forme d'un GFCDD ou, plus particulièrement, d'un GFCDD synchronisé par les événements d'entrée/sortie. Avant de générer l'implantation matérielle optimisée, il explore l'espace de solutions par l'intermédiaire de transformations spatio-temporelles appliquées à la spécification initiale (comme nous verrons dans le chapitre 6). Afin de trouver une implantation qui respecte les contraintes temporelles et qui minimise le nombre de ressources matérielles utilisées. À la fin de cette étape de synthèse comportementale, nous avons un graphe matériel d'implantation, obtenu par traduction directe du graphe algorithmique transformé, et un ensemble d'équations, obtenu par analyse des relations de voisinage entre les frontières de factorisation du graphe algorithmique. Cet ensemble d'équations va permettre de synthétiser la partie contrôle de l'implantation. La synchronisation entre les sommets du graphe matériel est assurée par un signal d'horloge. Après l'étape de synthèse, nous disposons d'un code VHDL synthétisable, représentant les opérateurs interconnectés et synchronisés par un signal d'horloge. Ce code VHDL sera fourni en entrée des outils de CAO qui, à leur tour, effectueront la synthèse logique de l'architecture-cible.

Pour représenter une implantation matérielle particulière, nous utilisons des graphes d'opérateurs interconnectés (graphe matériel), dont chaque sommet représente un opérateur et chaque arc une connexion inter-opérateurs. En plus de la partie chemin de données, composée des opérateurs et de ses interconnexions, déjà existante dans le graphe algorithmique, le graphe matériel contient une partie contrôle, composée d'unités de contrôle et des signaux de contrôle.

Comme nous verrons par la suite, cette approche de synthèse permet de synthétiser les boucles exprimées au niveau des frontières de factorisation sans être contraint de procéder à un déroulage (défactorisation), en fait, cette transformation n'est utilisée dans le cadre d'AAA que pour transformer le graphe d'algorithme initial afin de répondre à des besoins d'optimisation en vue de la satisfaction des contraintes temps réel et non pas pour des besoins de simplifier la synthèse. Elle permet aussi la synthèse du traitement conditionnel et le système de contrôle généré est un système délocalisé qui associe à chaque frontière sa propre unité de contrôle.

Nous décrivons dans ce qui suit succinctement l'ensemble des règles et transformations définies en vue de permettre la transformation du graphe d'algorithme initial (GFCDD) en un graphe matériel comprenant les chemins de données et de contrôle du circuit correspondant à l'implantation matérielle finale.

5.4.1 Synthèse du chemin de données

La première transformation en vue de l'implantation sur circuit d'une spécification algorithmique sous la forme d'un GFCDD est la traduction matérielle de ce graphe d'algorithme en un graphe d'implantation décrivant le chemin de données du circuit correspondant.

Principe

La synthèse du chemin de données consiste à faire correspondre, par traduction directe, à chaque sommet du graphe factorisé et conditionné de dépendances de données d'une spécification (graphe algorithmique G_{al}), un opérateur (composant d'une bibliothèque VHDL : combinatoire pour un sommet opération, multiplexeur et/ou registre pour un sommet frontière....) et, à chaque dépendance de données, une connexion interconnectant les ports reliant des opérateurs correspondants.

Le parallélisme potentiel, lié à l'ordre partiel établi par les dépendances de données entre opérations, devient ainsi un parallélisme effectif entre opérateurs.

Opérateurs matériels

Pour représenter l'implantation matérielle, nous utilisons des graphes d'opérateurs interconnectés (graphe matériel), dont chaque sommet représente un opérateur et chaque arc une connexion inter-opérateurs. L'étiquetage des sommets et des arcs permet de les caractériser. Chaque sommet représentant un opérateur reçoit une étiquette comportant : nom, surface et latence de l'opérateur, et chaque arc est étiqueté avec la durée de communication entre les opérateurs. Ces opérateurs matériels peuvent être soit un composant d'une bibliothèque VHDL, combinatoire et/ou séquentiel pour un sommet opération de calcul, soit un multiplexeur pour le sommet de factorisation Fork, démultiplexeur pour le sommet de factorisation Join, registre pour le sommet Iterate I, encodeur de priorité et multiplexeur pour un sommet Select.

On va donc présenter pour les principaux types de sommets algorithmiques les opérateurs matériels correspondants. Pour les distinguer graphiquement les premiers auront une forme ronde les secondes rectangulaires.

Opérateurs de base

L'opérateur Calcul Il permet l'implantation du sommet calcul correspondant sous la forme d'un circuit combinatoire ou séquentiel donné. Il s'agit donc selon le cas, soit d'un opérateur logique (and, or,..) arithmétique (additionneur, multiplieur,...) ou de calculs plus complexes définis par l'utilisateur selon ses besoins.

La figure 5.2 montre deux exemples d'utilisation de l'opérateur *CALCUL* : (a) le demi-additionneur de deux signaux e_1 et e_2 ; et (b) le filtre moyenneur appliqué sur une image I en produisant l'image filtrée I' en sortie.

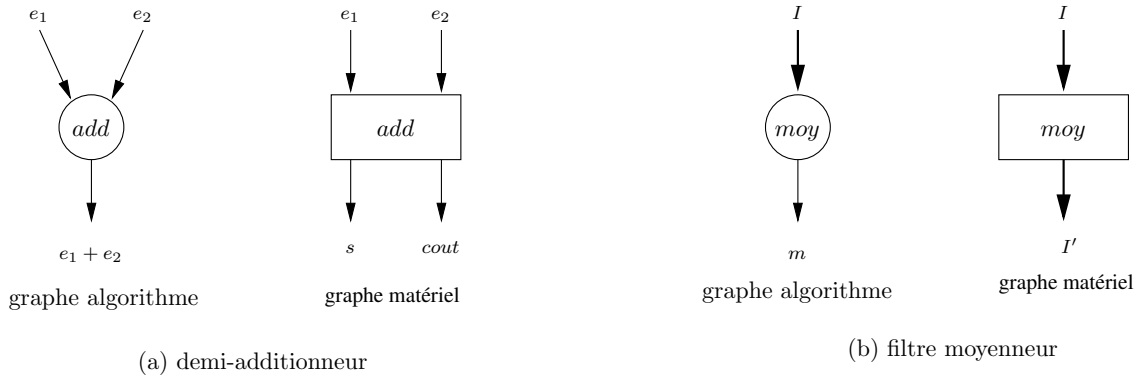


FIG. 5.2 – Exemples d'opérateurs *CALCUL*

L'opérateur Implode Cet opérateur, identifié par M, ne réalise qu'une transformation de typage, en effectuant un regroupement ordonné de d bus unidirectionnels, comme le montre la figure 5.3. La transformation consiste à implanter des connexions directes entre les opérateurs en amont et l'opérateur en aval.

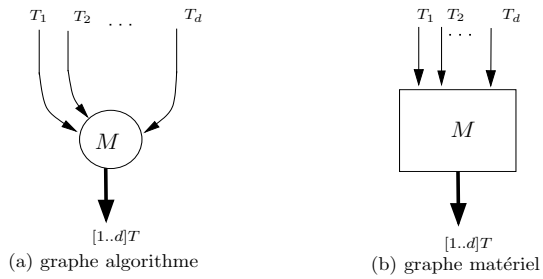
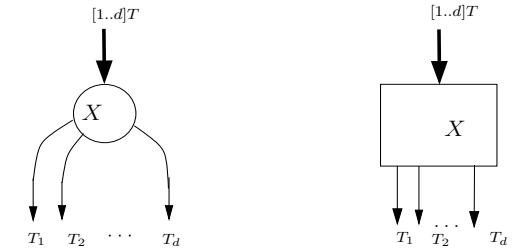


FIG. 5.3 – Opérateur Implode

L'opérateur Explode Cet opérateur, identifié par X, réalise une décomposition d'un bus unidirectionnel en d sous-bus, comme le montre la figure 5.4. La transformation consiste à implanter des connexions directes entre l'opérateur en amont et les opérateurs en aval.



(a) graphe algorithme (b) graphe matériel

FIG. 5.4 – Opérateur Explode

L'opérateur Retard Cet opérateur est un registre R qui maintient en sortie la valeur qu'il avait en entrée à la fin du cycle d'horloge précédent, pendant toute la durée du cycle d'horloge. Cet opérateur dispose en entrée : d'un signal d'horloge (*clk*), un signal de chargement (*load*), de la valeur d'initialisation (*init*) et la donnée d'entrée (e_i) (voir figure **??**).

Opérateurs de factorisation

L'opérateur Fork L'opérateur FORK doit intégrer tous les circuits nécessaires à l'implantation de la factorisation du flot de données en son entrée sous forme par exemple d'un vecteur T[1..d], en ses d éléments T_i en sortie. Selon l'implantation parallèle ou séquentielle de la factorisation du flot de données en entrée du sommet FORK, nous distinguons deux sommets opérateurs matériels correspondants, comme il est illustré sur la figure 5.5 :

- l'opérateur F_{par} permettant l'implantation parallèle en transformant la factorisation du flot de données au niveau algorithmique en une répétition spatiale au niveau matériel. Notons que l'implantation de l'opérateur F_{par} correspond dans ce cas particulier à son correspondant défactorisé, i.e l'opérateur Explode X ;
- l'opérateur F_{seq} permettant l'implantation séquentielle en transformant la factorisation du flot de données au niveau algorithmique en une itération (répétition temporelle) au niveau matériel. On remarque que l'opérateur *Fork* dispose en son entrée du signal valeur de comptage (*cpt*), du compteur associé à sa frontière de factorisation lui permettant le séquencement temporel des différentes itérations.

L'opérateur Join Cet opérateur intègre tous les circuits nécessaires à l'implantation de la factorisation des flots de données en son entrée, sous forme par exemple de d éléments, en un vecteur T[1..d] qu'il fournit en sortie.

Selon l'implantation parallèle ou séquentielle de la factorisation du flot de données en entrée du sommet JOIN, nous distinguons deux sommets opérateurs matériels correspondants, comme il est montré sur la figure 5.6 :

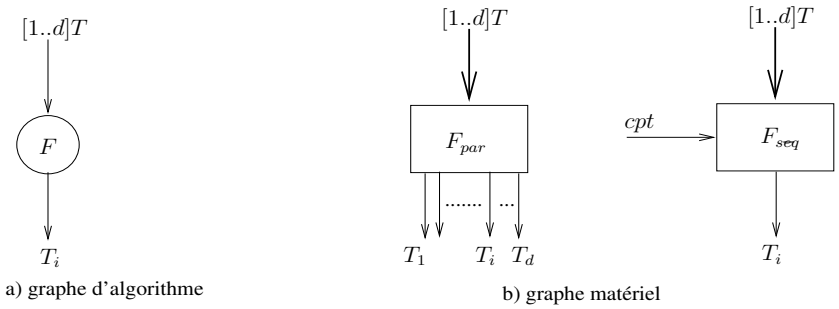


FIG. 5.5 – Opérateur Fork

- l'opérateur J_{par} permettant l'implantation parallèle en transformant la factorisation du flot de données au niveau algorithmique en une répétition spatiale au niveau matériel. Notons que l'implantation de l'opérateur J_{par} correspond, dans ce cas particulier, à son correspondant défactorisé i.e l'opérateur Implode M ;
- l'opérateur J_{seq} permettant l'implantation séquentielle en transformant la factorisation du flot de données au niveau algorithmique en une itération (répétition temporelle) au niveau matériel. On remarque que l'opérateur *Join* dispose en son entrée du signal valeur de comptage (*cpt*), reçu du compteur associé à sa frontière de factorisation lui permettant le séquencement temporel et cadencé par l'horloge d'entrée *clk*.

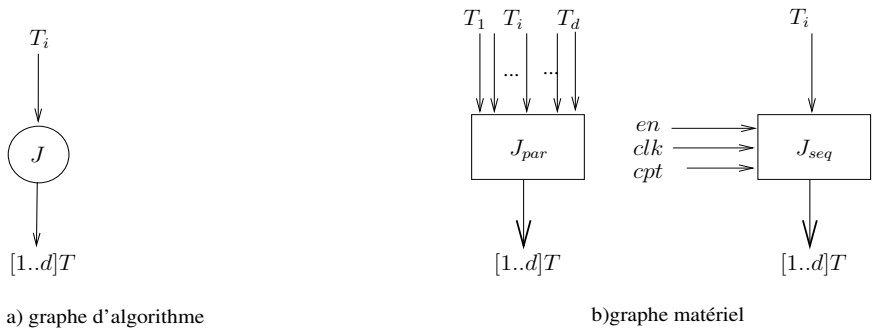
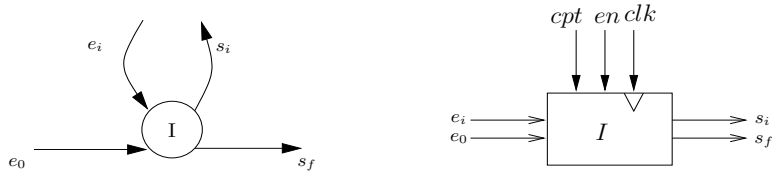


FIG. 5.6 – Opérateur Join

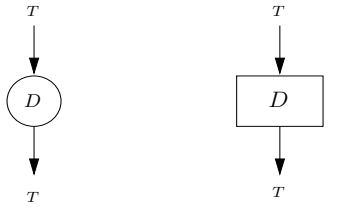
L'opérateur Iterate Cet opérateur, identifié par I, implante la factorisation des dépendances de données inter-motifs. Il dispose en entrée : d'un signal de validation (*en*) permettant la mémorisation de la valeur de l'itération précédente et d'un signal valeur de comptage périodique *cpt* cadencé par l'horloge d'entrée *clk*. On constate que les dépendances inter-motifs imposent un ordre total entre les répétitions factorisées et par conséquent une seule implantation de la factorisation associée qui est l'implantation séquentielle.



a) graphe d'algorithme b) graphe matériel

FIG. 5.7 – Opérateur Iterate

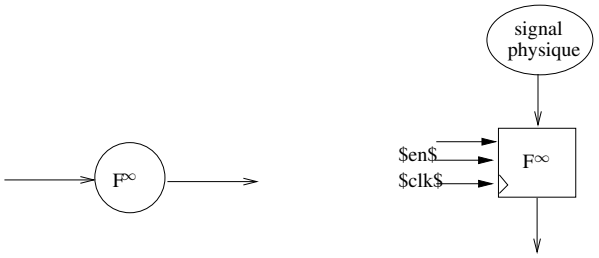
L'opérateur Diffusion Identifié par D, cet opérateur ne sert qu'à marquer des dépendances qui traversent une frontière de factorisation. Il implante donc des connexions directes entre son entrée et sa sortie. On remarque que l'opérateur *Diffusion* (figure 5.8) ne reçoit pas le signal valeur de comptage (cpt), parce qu'il fournit en sortie la valeur de son entrée pendant un cycle complet du compteur associé à sa frontière de factorisation. Il n'y a pas de séquencement temporel.



a)graphe d'algorithme b)graphe matériel

FIG. 5.8 – Opérateur Diffusion

L'opérateur Fork$^\infty$ (Capteur) L'opérateur capteur, identifié par F^∞, implante l'énumération d'un flot de données infini $[1..\infty]T'$ en entrée. Cet opérateur doit intégrer tous les circuits nécessaires à son fonctionnement et donc éventuellement, ceux qui assurent la conversion et l'échantillonnage des signaux reçus en entrée. Le contrôle de l'opérateur capteur est assuré par les signaux d'horloge (clk), de validation (en) et de remise à zéro (rst).



a) graphe d'algorithme b) graphe matériel

FIG. 5.9 – Opérateur Capteur

L'opérateur Join$^\infty$ (Actionneur) L'opérateur actionneur, identifié par J^∞, implante la collecte infinie des flots de données finis $[1..\infty]T'$ en son entrée. Cet opérateur doit intégrer tous les circuits nécessaires à son fonctionnement et donc éventuellement, ceux qui assurent la conversion des signaux numériques reçus en signaux analogiques en sortie. Le contrôle de l'opérateur actionneur est assuré par les signaux d'horloge (clk), de validation (en) et de remise à zéro (rst).

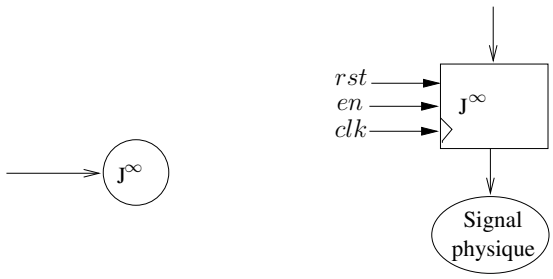


FIG. 5.10 – Opérateur Actionneur

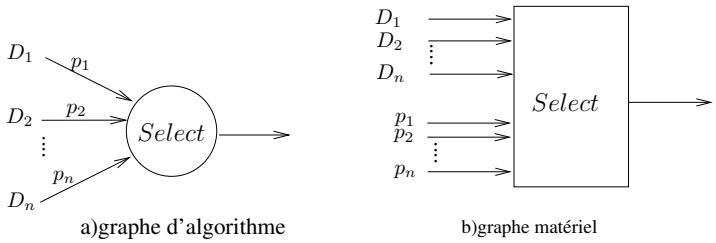


FIG. 5.11 – Opérateur Select

Opérateurs de conditionnement

L'opérateur Select Cet opérateur (figure 5.11)permet d'acheminer vers sa sortie l'entrée disponible D_i ayant la plus haute priorité p_i. Il doit donc intégrer tous les circuits nécessaires à l'encodage de priorité et à la sélection de la donnée : encodeur de priorité et multiplexeur. Il dispose en son entrée des entrées associées aux signaux d'encodage des priorités et des entrées associées aux données issues des sous-graphes conditionnés.

Formalisation de la synthèse du chemin de donnée

Les règles de synthèse du chemin de données se réduisent à une simple transformation du graphe d'algorithme $G_{al}(O, D_d \cup D_d)$ en un graphe matériel représentant le chemin de données $G_{cd}(O''_1, D''_1)$. Elle fait correspondre :

- à chaque opération $o_i \in O$ l'opérateur matériel correspondant. Cet opérateur sera modélisé sur le graphe matériel par un sommet o_i" $\in O''$ étiqueté par sa latence et sa surface,

– à chaque dépendance de donnée $d_i \in D_d$ une connexion physique entre les opérateurs correspondants modélisée sur le graphe matériel du chemin de donnée G_{cd} par une dépendance $d"_i \in D"_{1_d}$.
– et à chaque dépendance de conditionnement $d_{i_c} = (o_i, \{o_{c_j}\}) \in D_c$ reliant l'opération d'évaluation de la condition o_i aux opérations conditionnées $\{o_{c_j}\}$, une connexion physique entre l'opérateur $o"_i$ implémentant l'opération o_i et l'opérateur Select $o"_{op_s}$ associé à cette condition.

En résumé cette transformation est la composition des deux applications suivantes :
– une application bijective Γ_2 de O sur $O"_1$ qui définit un isomorphisme de graphes entre le sous-graphe $G_{al} = (O, D_d)$ et $G_{cd} = (O"_1, D"_{1_d})$ telle que $D"_{1_d}$ est l'ensemble des arcs $(\Gamma_2(o_i), \Gamma_2(o_j))$ pour tous les arcs $d_i = (o_i, o_j) \in D_d$.

$$\Gamma_2 : \quad G_{al}(O, D_d) \quad \longrightarrow \quad G_m(O"_1, D"_{1_d}) \left\{ \begin{array}{lcl} o_i & \longmapsto & o"_{op_i} \\ d_i = (o_i, \{o_{k(i)}\}) & \longmapsto & d"_i = (\Gamma_2(o_i), \{\Gamma_2(o_{k(i)})\}) \end{array} \right.$$

– et une application Γ'_2 de $G_{al} = (O, D_d \cup D_c)$ vers $G_{cd} = (O"_1, D"_{1_c})$ qui pour chaque $d_{i_c} = (o_i, \{o_{i_c}\}) \in D_c$ détermine le chemin ch reliant l'opération conditionnée o_{i_c} et l'opération Select o_{S_i} associée à la condition calculée par o_i puis crée la dépendance associée sur le graphe matériel entre $o"_{op_i}$ et $o"_{op_{S_i}}$ (chemin déterminé par fermeture transitive).

Le graphe $G_{cd} = (O"_1, D"_{1_d} \cup D"_{1_c})$ décrit en termes d'opérateurs et de leurs interconnexions constitue le chemin de données du circuit à synthétiser. L'algorithme 1 utilise les transformations Γ_2 *et* Γ'_2 pour construire ce graphe chemin de données à partir du graphe d'algorithme de l'application.

$$G_{al} = (O, D) \stackrel{Algorithm1}{\longrightarrow} G_{cd} = (O"_1, D"_1)$$

$$\left. \begin{array}{ccc} G_{al}(O, D_d) & \stackrel{\Gamma_2}{\rightarrow} & G_{cd}(O"_1, D"_{1_d}) \\ G_{al}(O, D_d \cup D_c) & \stackrel{\Gamma'_2}{\rightarrow} & G_{cd}(O"_1, D"_{1_c}) \end{array} \right\} \stackrel{\cup}{\rightarrow} G_{cd} = (O"_1, D"_1) \; avec \; D"_1 = D"_{1_d} \cup D"_{1_c}$$

Ainsi, après avoir fait les transformations Γ_2 *et* Γ'_2 on fait l'union des dépendances des deux graphes obtenus.

Algorithm 1 Synthèse du *chemin de données*

Entrée : Le graphe $G_{al} = (O, D = D_d \cup D_c)$, les transformations Γ_2, Γ'_2
Sortie : Graphe matériel du chemin de donnée $G_{cd} = (O"_1, D"_1)$
Begin
$O"_1 \leftarrow \phi$; $D"_1 \leftarrow \phi$; {initialisation}
for all $o_i \in O$ **do**
  Définir et ajouter $o"_{op_i} = \Gamma_2(o_i)$ dans $O"_1$;
end for
for all $d_i = (o_i, \{o_{k(i)}\}) \in D_d$ **do**
  Définir et ajouter $(\Gamma_2(o_i), \{\Gamma_2(o_{k(i)})\})$ *dans* $D"_{1_d}$;
end for
for all $d_{i_c} = (o_i, o_{i_c}) \in D_c$ **do**
  Déterminer le sommet select o_{S_i} correspondant par fermeture transitive ;
  Ajouter l'arc $(o"_i, o"_{op_{S_i}})$ *dans* $D"_{1_c}$; {avec $o"_{op_{S_i}} = \Gamma_2(o_{S_i})$}
end for

5.4.2 Synthèse du chemin de contrôle

Le circuit correspondant à l'implantation matérielle comprend le chemin de données auquel on associe un chemin de contrôle afin de gérer l'ordre d'exécution des opérateurs du chemin de données.

Principe

Le chemin de contrôle consiste à produire l'ensemble des signaux de synchronisation permettant de contrôler le transfert des données entre les opérateurs ainsi que leur ordre d'exécution. Cette synchronisation gère les transferts entre les registres des opérateurs. Pour qu'un registre puisse transiter, deux conditions sont nécessaires : les nouvelles données en amont du registre doivent être stables, et les consommateurs en aval du registre doivent avoir fini de consommer les données précédentes.

Si de plus les données en amont d'un opérateur X comprenant des registres viennent de différentes sources avec des durées de propagation différentes, il est nécessaire d'avoir un opérateur synchronisé. La synchronisation d'un opérateur X est possible à travers l'utilisation d'un protocole de communication de type requête/acquittement [], comme le montre la Fig. 5.12. Les producteurs des données en amont de X (Td_1 et Td_2 sur la Fig. 5.12, le suffixe d signifie aval - *downstream*) indiquent la disponibilité de ces données en provoquant une transition des signaux de requête (rd_1 et rd_2). L'opérateur X utilise ces données seulement quand son entrée de requête (ru) est active. L'opérateur X indique à ses producteurs en amont qu'il a fini de consommer leurs données en activant sa sortie d'acquittement (au, le suffixe u signifie amont -*upstream*). Il en va de même symétriquement du côté aval de X.

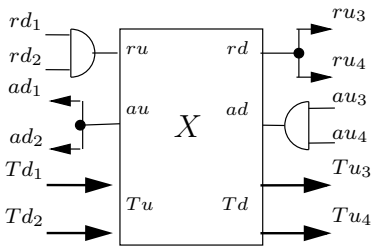


FIG. 5.12 – Opérateur synchronisé

Par conséquent, la synchronisation du circuit correspondant à l'implantation matérielle d'un algorithme spécifié sous la forme d'un graphe factorisé et conditionné de dépendance de données se résume ainsi à la synchronisation des signaux de requête et d'acquittement correspondant aux dépendances de données entre les opérateurs du chemin de données du circuit synthétisé.

Étant donné que ces opérateurs sont regroupés en frontières de factorisation et que leurs consommations et productions de données se font de manière synchrone au niveau de leurs frontières (i.e rythmées par la même horloge), le système de contrôle doit être un système local à chaque frontière où chaque frontière de factorisation disposera de sa propre logique de contrôle basée sur les relations de consommation/production entre les frontières.

Comme les dépendances de données ou encore de consommation/production entre les frontières de factorisation du graphe algorithmique impliquent au niveau matériel des dépendances de contrôle, l'analyse de ces relations permettra par la suite d'établir les relations de contrôle au niveau du graphe

d'implantation. Il est donc nécessaire de disposer d'une structure permettant de modéliser ces relations de consommation/production entre les frontières de factorisation de façon explicite.

Pour ce faire, ces relations de dépendances de données entre frontières appelées *relations de voisinage entre frontières* seront représentées sous la forme d'un graphe, appelé *graphe de voisinage* G_v, dont les sommets représentent les frontières de factorisation et dont les arcs orientés représentent les transferts de données entre les opérateurs frontières à travers des opérateurs de calcul et/ou de communication. L'orientation de l'arc indique la relation de production/consommation des données : "producteur" → "consommateur". Comme nous verrons plus loin (section 5.4.2), ce graphe intermédiaire résultat d'une première transformation du graphe algorithmique permettra d'établir les relations de contrôle de manière simple et systématique en suivant des règles simples de synthèse basées sur le modèle RTL et des mécanismes de transferts de données synchronisés.

Graphe de voisinage

Il est tout d'abord nécessaire de préciser le sens que l'on donnera à la notion de relation de voisinage entre frontières de factorisation : d'un point de vue comportemental ou opératoire, chaque frontière peut être consommatrice (située en aval) ou/et productrice (située en amont) par rapport à une autre frontière, en fonction des dépendances de données qui existent entre elles. Deux frontières sont dites voisines s'il existe entre elles au moins une relation de dépendance de donnée directe qui ne passe pas par l'intermédiaire d'une troisième frontière.

Ces relations de voisinage entre les frontières de factorisation du graphe d'algorithme G_{al} peuvent être décrite par un graphe G_v appelé graphe de voisinage. Les noeuds de ce graphe représentent les frontières de factorisation et les arcs orientés entrant et sortant des sommets représentent les transferts de données entre les frontières. L'orientation de l'arc indique les relations de production-consommation : l'arc part d'un producteur vers un consommateur. Comme ces relations de production/consommation de données peuvent elles même être conditionnées, on distingue ainsi les arcs de dépendances de production/consommation (traits continus), des arcs de dépendances de production/consommation conditionnés (traits en pointillé).

Il est intéressant de rappeler ici qu'une frontière de factorisation sépare deux zones, l'une interne "rapide" étant répétée par rapport à l'autre externe "lente". Ces côtés 'lent et rapide' de la frontière sont dûs à la variation de la fréquence des échanges des données au niveau des sommets de factorisation (deux frontières différentes seront rythmées par deux horloges différentes). Comme chaque frontière peut aussi être consommatrice (située en aval) ou/et productrice (située en amont) par rapport à une autre frontière, en fonction des dépendances de données qui existent entre elles, les sommets du graphe de voisinage représentant les frontières de factorisation peuvent être sous-divisés schématiquement en quatre régions correspondant aux consommations/productions des deux côtés : lent et rapide, comme le montre la Fig. 5.13.

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

Ce graphe de voisinage sera modélisé par $G_v = (O', V')$ où chaque sommet, o'_{F_i} de O' est un sommet frontière défini par un quadruplet $(SC_{F_i}, SP_{F_i}, FC_{F_i}, FP_{F_i})$ modélisant les quatre régions du sommet.

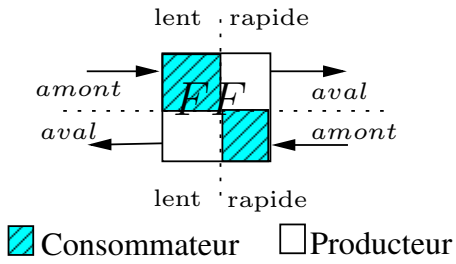


FIG. 5.13 – Sommet d'un graphe de voisinage représentant la frontière de factorisation FF

L'ensemble des arc incidents à SC_{F_i} définit l'ensemble des consommateurs lents de F_i, l'ensemble des arc incidents à SP_{F_i} définit l'ensemble des producteurs rapides de F_i, l'ensemble des arc incidents à FC_{F_i} définit l'ensemble des consommateurs rapides de F_i et l'ensemble des arc incidents à FP_{F_i} définit l'ensemble des producteurs rapides de F_i. Ce graphe de voisinage résultant des transformations faites sur l'hypergraphe orienté décrivant l'algorithme à implanter permettra par la suite d'établir les relations de contrôle au niveau du graphe d'implantation.

Construction du graphe de voisinage

Rappelons que l'appartenance à une même frontière de factorisation défini une relation d'équivalence $\Re$ sur O et que les classes d'équivalence correspondantes $O/\Re$, notées $\dot{F}_i$, définissent les frontières de factorisation. Ces frontières de factorisation modélisent ainsi les sommets du graphe de voisinage $G_v = (O', D')$ où $Card(O/\Re) = O'$.

On définit donc l'application Γ_1 *de* $O/\Re$ *vers* O' permettant la construction des sommets o'_{F_i} de O' comme suit :

$$\begin{array}{rccc} \Gamma_1 : & O/\Re & \longrightarrow & O' \\ & \dot{F}_i & \longmapsto & o'_{F_i} \end{array}$$

Notons aussi que la relation d'équivalence $\Re$ définie sur O répartit l'ensemble des dépendances $d_i \in D$ en dépendances *intra-classes* et *inter-classes* entre les différents $\dot{F}_i$:

- une dépendance d_j est dite dépendance *intra-classe* de $\dot{F}_i$ si : $\gamma^{-1}(d_j) \in \dot{F}_i \wedge \gamma(d_j) \cap \dot{F}_i \neq \phi$ où $\gamma^{-1}(d_j)$ défini l'émetteur de la dépendance d_j et $\gamma(d_j)$ le(s) récepteurs de d_j.
- une dépendance d_j est dite *inter-classe* de $\dot{F}_i$ si : $(\gamma^{-1}(d_j) \in \dot{F}_i \wedge \gamma(d_j) \cap \dot{F}_i = \phi) \vee (\gamma^{-1}(d_j) \notin \dot{F}_i \wedge \gamma(d_j) \cap \dot{F}_i \neq \phi)$.

Ces dépendances permettent de déduire les relations de consommation/production des données entres frontières et par conséquent l'ensemble des arcs du graphe de voisinage.

L'ensemble des dépendances intra-classes de chaque frontière F_i détermine l'ensemble $D_{IN}(\dot{F}_i)$ à partir duquel on calcule le chemin de données le plus long ayant comme source et puits des sommets de factorisation de F_i, si de tels chemins existent, alors la frontière est elle-même productrice et consommatrice par rapport à elle même ce qui définit dans G_v une boucle au niveau du sommet o'_{F_i}. Les côtés source et destination de cet arc seront déduits des côtés source et destination finale du chemin correspondant.

L'ensemble des dépendances inter-classes de chaque frontière F_i permet de déduire l'ensemble

$D_{OUT}(\dot{F}_i)$ contenant que les dépendances issues de F_i à partir desquels on détermine l'ensemble des chemins de données les plus longs ayant comme source un sommet de factorisation de la frontière F_i elle même ($o^{l/r,F_i}$) et comme puits un sommet de factorisation $o^{l/r,F_j}$ d'une autre frontière F_j ($j \neq i$). Pour chaque chemin reliant F_i et F_j dans G_{al}, on définit une dépendance entre o'_{F_i} et o'_{F_j} dans le graphe de voisinage $G_v = (O', D')$ si une telle dépendance existe déjà, on ne la rajoute pas dans D'.

Pour déterminer la région du sommet du graphe de voisinage à laquelle sera reliée la dépendance de donnée précédemment définie, on récupère du chemin correspondant l'abcisse du couple étiquetant la première dépendance du chemin ainsi que la coordonnée du couple étiquetant la dernière dépendance du chemin. Ces deux éléments déterminent ainsi les côtés source et destination de la dépendance de donnée dans le graphe de voisinage G_v.

$$\{(O_{i_1}\underbrace{(s,f)O_{i_2}), (O_{i_2}(f,f)O_{i_3}), ...(O_{i_k-1}(s,f)}\,O_{i_k})\} \overset{Correspondance}{\longrightarrow} (O_{i_1}(s,f)O_{i_k})$$

La construction d'un graphe de voisinage à partir du graphe algorithmique est réalisée par l'algorithme 2 page 99 :

$$G_{al} = (O, D) \overset{Algorithm2}{\longrightarrow} G_v = (O', D')$$

Ce dernier cherche d'abord à déterminer l'ensemble des dépendances internes à chaque frontière $D_{IN}(\dot{F}_i)$ à partir duquel il recherche les chemins reliant deux sommets de factorisation quelconques de la frontière en question. De tels chemins créent des arcs de voisinage entre le sommet frontière lui-même dans le graphe de voisinage G_v. L'algorithme cherche ensuite à déterminer l'ensemble des dépendances externes à la frontière $D_{OUT}(\dot{F}_i)$ à partir duquel il recherche les chemins reliant un sommet de factorisation de la frontière en question à un autre sommet de factorisation d'une autre frontière. De tels chemins permettent de déterminer et de créer des arcs de voisinage entre les deux sommets frontières correspondant dans le graphe de voisinage G_v.

On reprend l'exemple d'algorithme applicatif traité au chapitre 4 (figure.4.13) et on représente cette fois-ci les limites de chaque frontière par une ligne droite pointillée sur laquelle on regroupe ses sommets de factorisation. Cette réorganisation du graphe de départ permet de mieux mettre en évidence les relations de voisinage entre les frontières et leurs dépendances de consommation/production.

En appliquant l'algorithme de construction du graphe de voisinage sur cet exemple, nous distinguons trois classes d'équivalences différentes correspondant chacune à une frontière que nous délimitons sur la figure Fig. 5.14 par des rectangles en traits pointillés :
$\dot{F}_1 = \{F_M^\infty, F_V^\infty, F_C^\infty, J_S^\infty, id, Select, = 1\}$
$\dot{F}_2 = \{F_2, D_2, J_2\}$
$\dot{F}_3 = \{F_{31}, F_{32}, I_3, mul, add\}$

À chaque ensemble $\dot{F}_i$ nous associons un sommet o'_{F_i} dans G_v ainsi O' sera égal à $\{o'_{F_1}, o'_{F_2}, o'_{F_3}\}$. Le graphe de voisinage G_v ainsi construit aura pour sommets $o'_{F_1}, o'_{F_2}, o'_{F_3}$.

Pour chaque ensemble $\dot{F}_i$ nous déterminons les ensembles des dépendances inter-classes $D_{OUT}(F_i)$ et intra-classes $D_{IN}(F_i)$ à partir desquels nous procédons au calcul des ensembles des chemins $CH_{out}(F_i)$ et $CH_{in}(F_i)$. Ainsi :

Pour F_1 **:** $D_{OUT}(F_1) = \{(F_M^\infty, F_2), (F_V^\infty, D_2)\}$
$D_{IN}(F_1) = \{(F_C^\infty, = 1), (= 1, id), (F_V^\infty, id), (id, Select), (Select, J_S^\infty)\}$

Algorithm 2 Construction du *graphe de voisinage*

Entrée : Le graphe $G_{al} = (O, D)$, la relation $\Re$
Sortie : Graphe de voisinage $G_v = (O', D')$
Begin
$O' \leftarrow \phi$; $D' \leftarrow \phi$; {initialisation}
Calculer l'ensemble $O/\Re$;
for all $\dot{F}_i \in O/\Re$ **do**
  Définir et ajouter o'_{f_i} dans O'
end for
for all $\dot{F}_i \in O/R$ **do**
  Calculer $D_{IN}(\dot{F}_i)$ tel que : $D_{IN}(\dot{F}_i) = \left\{ d_j \in D : \gamma^{-1}(d_j) \in \dot{F}_i \wedge \gamma(d_j) \cap \dot{F}_i \neq \phi \right\}$
  Déterminer à partir de $D_{IN}(\dot{F}_i)$ l'ensemble des chemins les plus longs $CH_{in}(F_i)$ ayant pour extremités des opérations de factorisation.
  if $CH_{in}(F_i) \neq \phi$ **then**
    $D' \longleftarrow D' \cup (o'_{f_i}, o'_{f_i})$
  end if
  Calculer $D_{OUT}(\dot{F}_i)$ tel que : $D_{OUT}(\dot{F}_i) = \left\{ d_j \in D : (\gamma^{-1}(d_j) \in \dot{F}_i \wedge \gamma(d_j) \cap \dot{F}_{i'} \neq \phi) \right\}$
  Déterminer à partir de $D_{OUT}(\dot{F}_i)$ l'ensemble des chemins les plus longs $CH_{out}(F_i)$ ayant pour source des opérations de factorisation de F_i (i.e de type $o^{l/r,F_i}$) et pour puit des opérations de factorisation de F_j (i.e de type $o^{l/r,F_j}$ avec $i \neq j$)
  for all $ch_{ij} \in CH_{out}(F_i)$ **do**
    définir $d' = (o'_{f_i}, o'_{f_j})$
    if $d' \notin D'$ **then**
      $D' \longleftarrow D' \cup d'$
    end if
  end for
end for
End

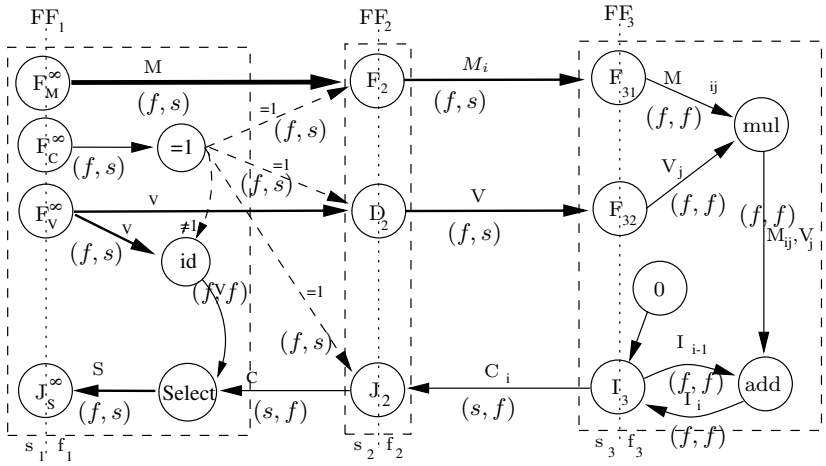


FIG. 5.14 – Spécification algorithmique factorisée du C-PMV

$CH_{OUT}(F_1) = \{\{(F_M^\infty(f,s)F_2)\}, \{(F_V^\infty(f,s)D_2)\}\}$
$CH_{IN}(F_1) = \{\{(F_V^\infty(f,f)id), (id(f,f)Select), (Select(f,f)J_S^\infty)\}\}$
À ce niveau $D' \longleftarrow \Phi \cup \{(o'_{F_1}(f,s)o'_{F_2}), (o'_{F_1}(f,f)o'_{F_1})\}$.
ainsi FF_1 est productrice de son côté rapide soit vers le côté lent de la frontière conditionnée FF_2 (arcs pointillés M et V), soit vers son côté lent "lui-même" (arc pointillé V).

Pour F_2 **:** $D_{OUT}(F_2) = \{(F_2, F_{31}), (D_2, F_{32}), (J_2, Select)\}$
$D_{IN}(F_2) = \Phi$
$CH_{OUT}(F_2) = \{\{(F_2(f,s)F_{31})\}, \{(D_2(f,s)F_{32})\}, \{(J_2(s,f)Select), (Select(f,f)J_S^\infty)\}\}$
$CH_{IN}(F_2) = \Phi$
À ce niveau $D' \longleftarrow D' \cup \{(o'_{F_2}(f,s)o'_{F_3}), (o'_{F_2}(s,f)o'_{F_1})\}$.
ainsi FF_2 est à son tour productrice soit de son côté rapide (arcs M_i et V) vers le côté lent de FF_3, soit de son côté lent (arcs C) vers FF_1.

Pour F_3 **:** $D_{OUT}(F_3) = \{(I_3, I_2)\}$
$D_{IN}(F_3) = \{(F_{31}, mul), (F_{32}, mul), (mul, add), (add, I_{31}), (add, I_{31}), (0, I_{31})\}$
$CH_{OUT}(F_3) = \{\{(I_3(s,f)I_2)\}\}$
$CH_{IN}(F_3) = \{\{(F_{31}(f,f)mul), (mul(f,f)add), (add(f,f)I_{31})\},$
$\{(F_{32}(f,f)mul), (mul(f,f)add), (add(f,f)I_{31})\}\}$
À ce niveau $D' \longleftarrow D' \cup \{(o'_{F_3}(s,f)o'_{F_2}), (o'_{F_3}(f,f)o'_{F_3})\}$.
ainsi FF_3 est à son tour productrice soit de son côté lent (arcs C_i) vers le côté rapide de FF_2, soit de son côté rapide (arcs mul, add) vers son côté rapide "lui-même".

Le graphe de voisinage ainsi construit est montré sur la figure Fig.5.15.

On constate que la frontière FF_1 est une frontière infinie. Elle n'a pas de voisines de son côté "lent" (car celui-ci correspond à l'environnement physique). FF_1 est à la fois productrice, soit par rapport à la frontière conditionnée FF_2 (arcs pointillés M et V), soit par rapport à elle même (arc pointillé V) et consommatrice, soit par rapport à FF_2 (arc pointillé C), soit par rapport à elle même (arc pointillé

V). FF_2 est à son tour aussi productrice (arcs M_i et V) et consommatrice (arc S_i) par rapport à FF_3. FF_3 est productrice et consommatrice par rapport à elle-même à travers les opérations de calcul *mul* et *add*. Chaque arcs reliant deux frontières sur le graphe de voisinage est déduit des dépendances de données entre ces frontières(D_{OUT}), il part du sommet frontière productrice vers le sommet frontière consommatrice. Cependant les arcs reliant le sommet frontière à lui même sont déduits des chemins de dépendances de données à l'intérieure de cette frontière (D_{IN}).

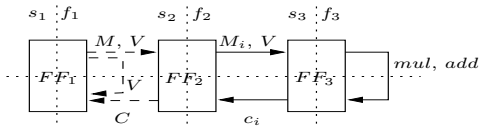


FIG. 5.15 – Graphe de voisinage du C-PMV

Après avoir précisé le processus de construction du graphe de voisinage sur lequel est fondé notre approche de génération du contrôle nous continuons à exposer cette approche dans ce qui suit.

Unité de contrôle

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

Nous associons ainsi à chaque frontière un compteur qui doit assurer le séquencement temporel nécessaire à l'implantation séquentielle de ses différents opérateurs de factorisations (F, J et I) pour séquencer leurs données d'entrée et/ou de sortie.

Ce séquencement temporel demande l'utilisation d'une horloge pour piloter les registres associés aux compteurs et aux opérateurs de factorisation (J, I, F^∞, J^∞ et I^∞) et d'une stratégie de contrôle pour coordonner les transitions d'état de ces registres.

Comme on associe à chaque frontière un compteur différent, les dépendances de données inter-frontières impliquent des dépendances de contrôle inter-compteurs. Le rôle du contrôle est donc d'expliciter, d'une part, les rapports entre le compteur et les opérateurs de factorisation d'une même frontière de factorisation et, d'autre part, les relations entre les compteurs associés aux différentes frontières de factorisation. La partie contrôle d'une implantation matérielle correspond aux compteurs et à la logique ajoutée pour piloter les compteurs, afin de coordonner leur opération.

Le système de contrôle proposé est un système de contrôle local, à chaque frontière. Où chaque frontière de factorisation disposera d'une **unité de contrôle** (**UC**) qui se chargera du bon déroulement de la factorisation interne à la frontière (i.e le séquencement des opérations internes à la frontière), et qui s'occupera aussi de ses communications (i.e relations de production/consommation) avec les frontières voisines en assurant la gestion des différents signaux de requête et d'acquittement amont et aval de ses différents opérateurs de factorisation. L'unité de contrôle est par conséquent composée d'un compteur C à d états (d étant le facteur de factorisation de la frontière associée) permettant de décompter les différents cycles de la factorisation et d'une logique supplémentaire afin de générer le protocole de communication entre frontières (signaux de requête et d'acquittement lent et rapide, en amont et en aval).

L'interface de l'unité de contrôle (Fig. 5.16) montre l'ensemble des signaux entrant et sortant de l'UC.

La valeur du compteur (*cpt*) commande les opérateurs de factorisation alors que le signal de validation *en* détermine les cycles d'horloges où les registres de ces opérateurs doivent transiter. Pour simplifier la description des ces différents signaux d'entrée et sortie nous adoptons par la suite la notation suivante :

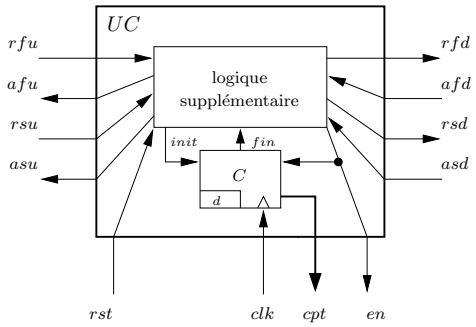


FIG. 5.16 – Unité de contrôle

s/f : *slow*/*fast* (lent/rapide),
u/d : *upstream*/*downstream* (amont/aval),
r/a : requête/acquittement,

- rfu : signal de requête rapide en amont (entrée),
- rfd : signal de requête rapide en aval (sortie),
- afd : signal d'acquittement rapide en aval (entrée),
- afu : signal d'acquittement rapide en amont (sortie),
- rsu : signal de requête lent en amont (entrée),
- rsd : signal de requête lent en aval (sortie),
- asd : signal d'acquittement lent en aval (entrée),
- asu : signal d'acquittement lent en amont (sortie),
- *cpt* : valeur de comptage (sortie),
- *en* : signal de validation compteur (sortie),
- *clk* : signal d'horloge (entrée),
- *rst* : signal de reset global (entrée).

Outre ces signaux d'entrée sortie, on retrouve aussi le signal interne *init* qui permet de remettre le compteur à son état initial 0 et le signal fin qui indique que le compteur est dans son état final $(d-1)$, et que la factorisation est donc terminée. Chaque unité de contrôle sera représentée dans le graphe matériel d'implantation par un sommet qui spécifie ses différents signaux d'entrée sortie. La figure 5.17 représente un exemple de sommet unité de contrôle UC_0.

Interconnexion des unités de contrôle : génération du contrôle

On rappelle que le principe de fonctionnement de ce système de contrôle est basé sur les relations producteur-consommateur entre frontières qui sont implantées par un jeu de requête/acquittement. Quand une frontière vue comme "consommatrice" a fini son calcul, elle envoie un signal de requête à

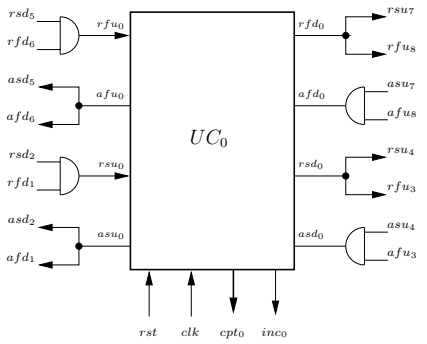


FIG. 5.17 – Sommet unité de contrôle

ses frontières "consommatrices". Une fois qu'une frontière "consommatrice" a reçu tous les signaux de requête de ses frontières "productrices", elle peut commencer son cycle de factorisation. Une fois le calcul terminé, elle envoie un signal d'acquittement à ses frontières "productrices" pour leur indiquer que les données ont été consommées. Quand tous les signaux d'acquittement ont été reçus, l'unité de contrôle d'une frontière "productrice" remet le signal de requête à zéro et la frontière reprend un nouveau cycle de factorisation. La figure 5.18 illustre ce principe.

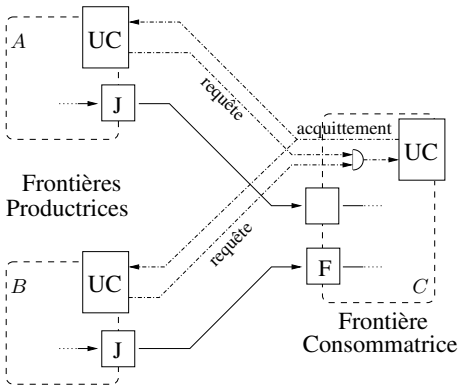


FIG. 5.18 – Principe du système de contrôle

Les unités de contrôle des différentes frontières peuvent être interconnectées de façon automatique à partir du graphe de relations de voisinage entre les frontières construit à partir du graphe d'algorithme de l'application à implanter. Dans ce graphe, les sommets correspondent aux unités de contrôle et les arcs correspondent aux signaux de requête transmis entre les unités de contrôle. Les signaux d'acquittement associés aux signaux de requête sont transmis entre les mêmes unités de contrôle, en sens inverse des signaux de requête. L'ensemble des unités de contrôle et leurs interconnexions constitue le chemin de contrôle de l'implantation.

Le processus de déduction du graphe chemin de contrôle se réduit ainsi à une simple transformation du graphe de voisinage $G_v(O', D')$ construit précédemment en un graphe d'interconnexion d'unités de contrôle $G_{cc}(O''_2, D''_2)$ (cc pour chemin de contrôle).

Initialement, on fait correspondre à chaque sommet frontière $o'_{F_i} \in O'$ du graphe de voisinage G_v, le sommet unité de contrôle correspondant $o"_{UC_i} \in O''_2$ du graphe chemin de contrôle en cours de construction. Puis à chaque arc orienté $d_{F_i,F_j} = (o'_{F_i}, o'_{F_j}) \in D'$ de G_v on associe un signal de requête transmis entre les sommets unités de contrôle correspondants ($o"_{UC_i}$,$o"_{UC_j}$) modélisé sur le graphe matériel G_{cc} par une dépendance $d^{rd-ru}_{UC_i,UC_j} \in D''_2$. Enfin, pour chaque dépendance $d^{rd-ru}_{UC_i,UC_j} \in D''_2$ ainsi créé, on ajoute une dépendance dans la direction inverse qui correspond au signal d'acquittement associé transmis, en sens inverse, entre les mêmes sommets unités de contrôle.

Quand plusieurs dépendances (signaux) arrivent à une même entrée d'un sommet unité de contrôle $o"_{UC_i}$, on en prend la conjonction par une porte logique ET. Si de plus ces dépendances correspondent à des dépendances conditionnées, elles seront d'abord acheminées vers un multiplexeur piloté par l'encodeur de priorité du sommet Select correspondant ce qui permettra d'acheminer en sortie le signal de requête ayant la plus haute priorité.

En résumé cette transformation est la composition de :

– Une bijection Γ_3 de O sur $O"_2$ qui définit un isomorphisme de graphes entre $G_v = (O', D')$ et $G_{cc} = (O"_2, D"_2)$ comme suit :

$$\Gamma_3 : \quad G_v(O', D') \quad \rightarrow \quad G_{cc}(O"_2, D"_2) \begin{cases} o'_{F_i} & \longmapsto & o"_{UC_i} \\ d' = (o'_{F_i}, \{o'_{F_{k(i)}}\}) & \longmapsto & d"^{rd-ru}_{UC_i,UC_{k(i)}} = (o"_{UC_i}, \{o"_{UC_{k(i)}}\}) \\ & avec & o"_{UC_l} = \Gamma_3(o'_{F_l}) \;\; l = i, k(i) \end{cases}$$

– Et d'une application Γ'_3 définie sur G_{cc} lui-même qui à chaque dépendance dans $D"_2$ définit et ajoute une dépendance dans la direction inverse afin de modéliser le signal d'acquittement correspondant au signal de requête.

$$\begin{array}{rcl} \Gamma'_3 : \quad G_{cc}(O"_2, D"_2) & \longrightarrow & G_{cc}(O"_2, D"_2) \\ d"^{rd-ru}_{UC_i,UC_j} = (o"_{UC_i}, o"_{UC_j}) & \longmapsto & d"^{ad-au}_{UC_j,UC_i} = (o"_{UC_j}, o"_{UC_i}) \end{array}$$

Ce nouveau graphe G_{cc} décrit en termes d'unités de contrôle et leurs interconnexions constitue le chemin de contrôle du circuit à synthétiser. L'algorithme 5 décrit sous forme algorithmique la démarche suivie par la composition des deux transformations ($\Gamma'_3 \circ \Gamma_3$) pour construire ce graphe du chemin de contrôle à partir du graphe de voisinage.

$$G_v(O', D') \overset{Algorithm5}{\rightarrow} G_{cc}(O"_2, D"_2)$$

Enfin, le graphe d'implantation finale $G_{im} = (O", D")$ est obtenu par l'union du graphe matériel d'opérateurs $G_{cd} = (O"_1, D"_1)$ et du graphe de contrôle $G_{cc} = (O"_2, D"_2)$ augmenté des dépendances $D"_c$ reliant les signaux de contrôle en sortie de chaque sommet unité de contrôle aux entrées correspondantes des opérateurs de factorisation. Il s'agit des différents signaux de validation *en*, de comptage *cpt*,....

$$G_{im} = (O", D") \;\; ou \;\; O" = O"_1 U O"_2 \;\; et \;\; D" = D"_1 U D"_2 U D"_c$$

5.4.3 Synthèse du circuit d'exécution

L'architecture circuit d'une implantation indique comment le système sera réellement mis en oeuvre. La synthèse du circuit permet donc de valider l'approche d'implantation, de montrer son efficacité et

Algorithm 3 Synthèse du *chemin de contrôle*

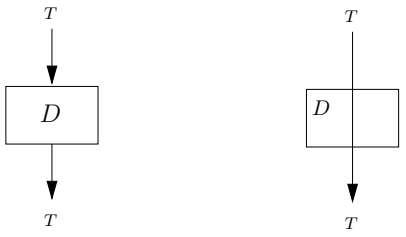
Entrée : Le graphe $G_v = (O', D')$, les transformations Γ_3, Γ'_3
Sortie : Graphe du contrôle $G_{cc} = (O"_2, D"_2)$
Begin
$O"_2 \leftarrow \phi$; $D"_2 \leftarrow \phi$; {initialisation}
for all $o_{F_i} \in O'$ **do**
  Définir et ajouter $o"_{UC_i} = \Gamma_3(o_{F_i})$ dans $O"_2$
end for
for all $d_i = (o_{F_i}, \{o_{F_{k(i)}}\}) \in D'$ **do**
  Définir et ajouter $(\Gamma_3(o_{F_i}), \{\Gamma_3(o_{F_{k(i)}})\})$ *dans* $D"_2$; {signal de requête $d^{rd-ru}_{UC_i,UC_{k(i)}}$}
end for
for all $d^{rd-ru}_{UC_i,UC_{k(i)}} = (o"_{UC_i}, \{o"_{UC_{k(i)}}\}) \in D"_2$ **do**
  Ajouter l'ensemble des arcs $d^{au-ad} = \Gamma'_3(o"_{UC_i}, \{o"_{UC_{k(i)}}\})$ *dans* $D"_2$; {signaux d'acquittement}
end for

aussi de bien la définir. Dès lors, pour valider notre modèle d'implantation nous avons proposé une correspondance circuit, c'est-à-dire un schéma électrique aussi bien pour les composants chemin de données (opérateurs) que pour les composants du chemin de contrôle (unité de contrôle).

Opérateurs matériels : chemin de données

Nous avons implémenté un certain nombre de sommets du GCFDD que nous regroupons dans la présentation suivante en différents types.

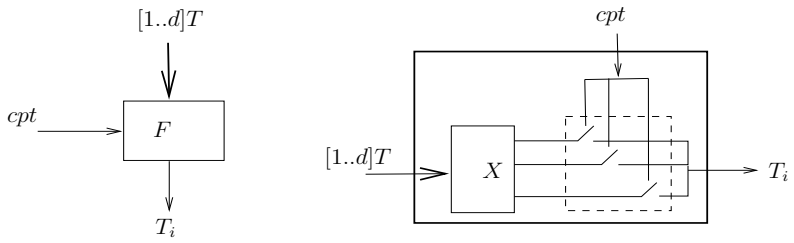
Opérateurs de factorisation Bien que les sommets frontières de factorisation ne jouent qu'un rôle "syntaxique" au niveau du graphe algorithmique consistant en la réduction de la taille des spécifications, les opérateurs matériels correspondants doivent réaliser le multiplexage/démultiplexage du groupes factorisés des données dans le cas de l'implantation séquentielle, sauf l'opérateur DIFFUSION, qui se contente de fournir la même valeur à chaque itération (voir figure5.19).



(a) graphe matériel : symbole (b) circuit : vue interne

FIG. 5.19 – L'opérateur *Diffuse*

L'opérateur FORK sur la figure 5.20 intègre :
- Un explode X, reçoit en entrée le signal T[1..d], vecteur composé de d signaux T_i qu'il fournit séparé en sortie,
- Un multiplexeur qui fournit les composantes du vecteur.



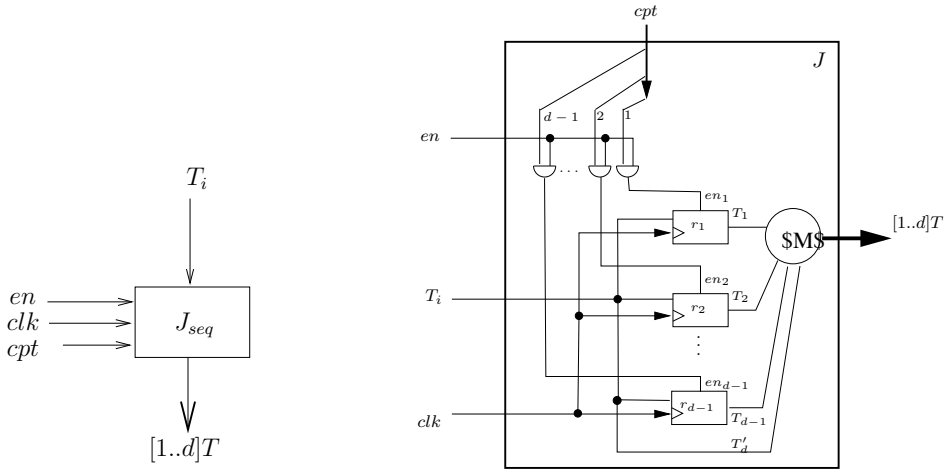
(a) graphe matériel : symbole (b) circuit : vue interne

FIG. 5.20 – L'opérateur *Fork*

Le correspondant séquentiel de l'opérateur JOIN (voir figure 5.21) doit intégrer :

– Un ensemble de $(d-1)$ registres, identifiés par $r_1, r_2, ..., r_{d-1}$, qui fournissent en sortie les signaux T[1..(d-1)]. Un registre r_i est validé par le signal en_i. Le signal en_i est produit si le signal *en* est actif et la valeur du signal *cpt* est correspondante. La dernière donnée T_d est utilisée durant le même cycle que son calcul et n'a donc pas besoin d'être sauvegardée dans un registre.
Ainsi, un signal T_i reçu en entrée est aiguillé sur l'un des $(d-1)$ registres, sélectionné par le signal en_i, pour i variant entre 1 et $(d-1)$. Tous les registres possèdent une entrée horloge (clk) et une entrée de validation (en) permettant de valider à l'instant donné l'écriture sur le registre sélectionné,

– Un implode, identifié par M, regroupe les d signaux T_i sous la forme d'un vecteur à d éléments en sortie.



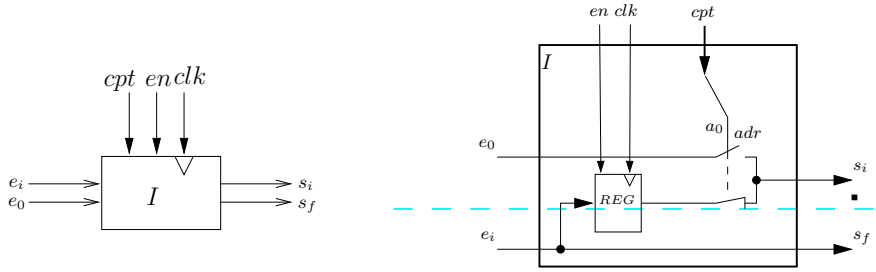
(a) graphe matériel : symbole (b) circuit : vue interne

FIG. 5.21 – L'opérateur *Join*

L'opérateur ITERATE sur la figure 5.22 intégre :

– Un registre, identifié par REG, qui mémorise la valeur de l'itération précédente quand le signal de validation *en* est actif.

– Un multiplexeur formé par deux interrupteurs contrôlés par *cpt* qui permet d'aiguiller la valeur d'initialisation (init), si le compteur est dans son état initial, sinon, il aiguille la sortie du registre REG



(a) graphe matériel : symbole

(b) circuit : vue interne

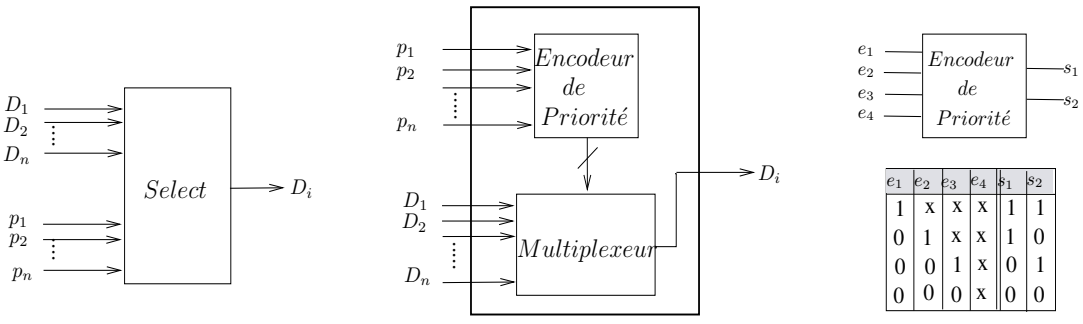
FIG. 5.22 – L'opérateur *Iterate*

Opérateurs de conditionnement : opérateur select

L'opérateur SELECT (figure 5.23) doit intégrer tous les circuits nécessaires à l'encodage de priorité et à la sélection de la donnée : encodeur de priorité et multiplexeur.

– Un encodeur de priorité générant à partir des 2^n valeurs de priorité reçues en entrée, le code sur n bits de l'entrée la plus prioritaire, ce code servira d'adresse de sélection au multiplexeur (voir la table de vérité donnée ci-dessous pour un encodeur de priorité à 4 entrées et 2 sorties).

– Un multiplexeur permet de fournir en sortie la donnée sélectionnée par l'adresse reçue de l'encodeur. Cette donnée est produite par l'opérateur conditionné ayant la plus haute priorité. L'entrée la moins prioritaire, sert à aiguiller la valeur par défaut, dans le cas où aucune condition n'est vraie.



e_1	e_2	e_3	e_4	s_1	s_2
1	x	x	x	1	1
0	1	x	x	1	0
0	0	1	x	0	1
0	0	0	x	0	0

(a) graphe matériel : symbole

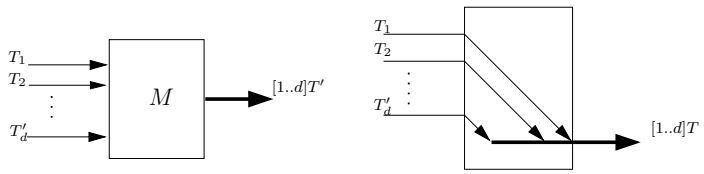
(b) circuit : vue interne

FIG. 5.23 – L'opérateur *Select*

Opérateurs de base

L'opérateur IMPLODE effectue le regroupement ordonné (dans l'ordre prédéfini trivial) de d données d'entrée de même type, en un vecteur de sortie de dimension d, c'est-à-dire l'élément disponible à l'entrée i sera l'i^{eme} élément du vecteur de sortie. Comme le montre la figure 5.24, cet opérateur ne réalise en

fait qu'une transformation de typage. L'implémentation de l'opérateur qui reçoit en entrée d données, codées sur n bits, correspond à $(d \times n)$ lignes de communication.



(a) graphe matériel : symbole (b) circuit : vue interne

FIG. 5.24 – L'opérateur *IMPLODE*

L'opérateur EXPLODE effectue la décomposition ordonnée (dans l'ordre prédéfini trivial) d'un vecteur de dimension d en ses d éléments, c'est à dire que l'i^{eme} élément du vecteur d'entrée sera disponible à la sortie i. Comme c'est le cas pour le sommet *IMPLODE*, le sommet *EXPLODE* ne correspond qu'à des lignes de communications.

Unité de contrôle : chemin de contrôle

Le circuit correspondant à l'unité de contrôle des frontières de factorisation finie est composé d'un compteur C, et d'une logique supplémentaire qui gère le protocole de contrôle des communications entre frontières (signaux de requête et d'acquittement lent et rapide, en amont et en aval), comme il est montré sur la figure 5.26. Une description détaillée de cette logique de contrôle et des équations des différents signaux d'entrée sortie cités ci-dessus, est décrite en détail dans [27].

Pour contrôler une frontière infinie, il est nécessaire d'utiliser une unité plus simple qui n'utilise pas de compteur, figure 5.25. On remarque notamment que les signaux rsu et asd sont systématiquement mis à 1, car il n'y pas de frontière en relation du coté lent. Par contre tous les autres signaux du coté rapide se comportent de manière analogue à ceux d'une UC de frontière finie.

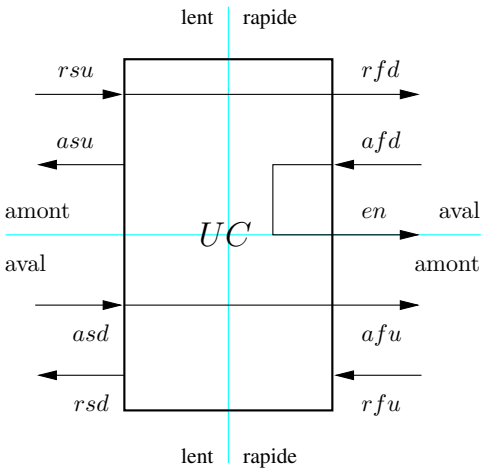


FIG. 5.25 – Unité de contrôle infinie

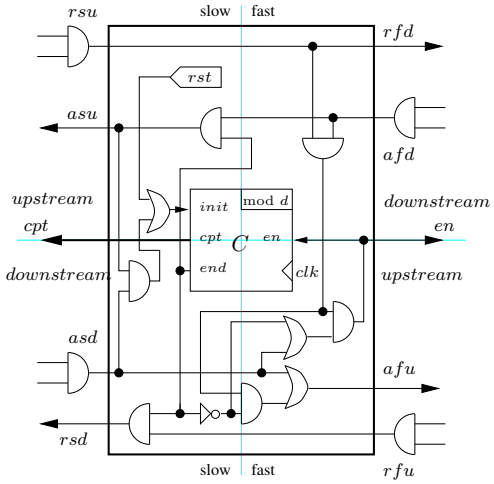


FIG. 5.26 – Unité de contrôle finie

5.5 Exemple d'implantation

La Fig. 5.27 représente l'implantation matérielle du C-PMV factorisé correspondant à la spécification algorithmique donnée dans Fig. 5.14. La partie du graphe Fig. 5.27.a) correspond au graphe matériel d'opérateurs $G_m = (O"_1, D"_1)$ constitué des opérateurs frontières de factorisation (F, D, J et I) et des opérateurs de calcul. La partie du graphe Fig. 5.27.b) correspond au graphe de contrôle $G_{cc} = (O"_2, D"_2)$ constitué par les sommets unités de contrôle UC_1, UC_2 et UC_3, et par leurs signaux de contrôle de requête 'r', et d'acquittement 'a'. Comme on peut le constater ce graphe est déduit automatiquement du graphe de voisinage G_v construit précédemment Fig. 5.15.

Les signaux *cpt*, *en* de chaque sommet unité de contrôle sont interconnectés aux sommets de factorisation de la frontière associée ($D"_c$) :

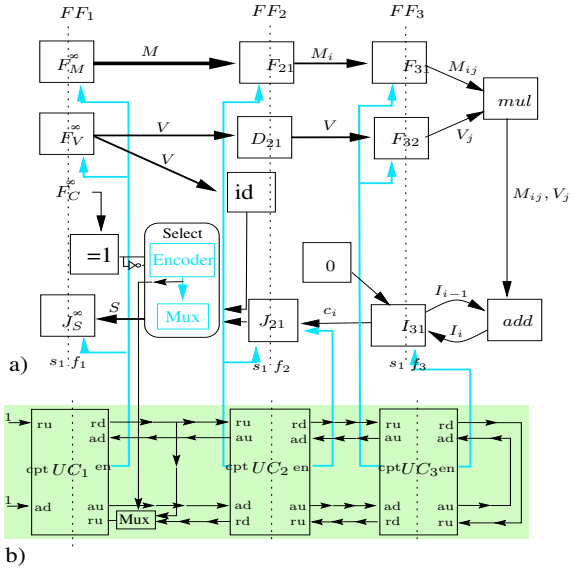


FIG. 5.27 – Implantation matérielle du PMV factorisé et conditionné

5.6 Conclusion

Dans ce chapitre, nous avons montré qu'en partant d'une spécification algorithmique basée sur un modèle de graphe de dépendances factorisé et conditionné de dépendances de données, il est possible d'obtenir une implantation matérielle sous la forme d'un circuit, en utilisant des règles simples de synthèse des chemins de données et de contrôle, formalisées en termes de transformations de graphe.

L'approche de synthèse préconisée concerne aussi bien le traitement conditionnel que les boucles finies sans recours à un déroulage préalable ce qui est le cas dans la plupart des outils de synthèse et génère un système de contrôle délocalisé permettant aux outils de CAO de placer les unités de contrôle le plus proches possible des opérateurs à contrôler.

Chapitre 6

Optimisation à l'aide d'heuristique

Après avoir présenté dans le chapitre précédent notre modèle d'implantation qui permet de construire une implantation matérielle valide à partir d'une description comportementale (spécification algorithmique), nous nous intéressons dans ce chapitre à une étape centrale du flot d'extension d'AAA qui est l'exploration de l'espace des solutions d'implantations possibles en vue de déterminer l'implantation optimisée. Le but est de permettre le choix de l'architecture qui satisfait d'une façon optimale les contraintes de l'application (fonctionnelles et non fonctionnelles). Pour cela, nous proposons un nouveau modèle d'estimation de performance au niveau algorithmique permettant une exploration rapide de l'espace d'architectures.

6.1 Introduction

La satisfaction des exigences des contraintes temps réel et d'embarquabilité souvent contradictoires et interdépendantes, impose la considération et l'expérimentation de plusieurs solutions avant de se décider sur la solution qui répond au mieux à l'ensemble des critères considérés. Dans un flot de conception complet, cette activité d'exploration de l'espace des solutions devient, de plus en plus, une composante primordiale en permettant d'apporter des solutions à des sous-problèmes interdépendants à différents points du cycle de "conception-implantation" : le partitionnement matériel/logiciel, l'allocation de ressources matérielles, la transformation de la spécification et l'estimation des performances temporelles et du nombre de ressources matérielles nécessaires à l'implantation [109]. Dans ce travail, nous ne nous intéressons qu'aux deux derniers sous-problèmes : la transformation de la spécification par l'intermédiaire d'une optimisation par défactorisation et la prédiction de la consommation de ressources matérielles et du temps de réponse de l'implantation optimisée, tout cela à partir du graphe algorithmique de l'application.

La recherche de la transformation qui respecte les contraintes temporelles, tout en minimisant l'augmentation du nombre de ressources matérielles nécessaires à l'implantation matérielle, demande d'examiner un nombre très important de combinaisons. Face à cette explosion combinatoire, il est nécessaire de réaliser cette exploration le plus rapidement possible afin que la solution soit obtenue dans un délai raisonnable à l'échelle humain.

En règle générale, l'efficacité de l'exploration de l'espace des solutions dépend d'une part de l'efficacité des techniques d'aide à l'évaluation/estimation rapide des performances et des coûts d'une implémentation éventuelle du système sans que celle-ci soit réalisée, ce qui rend possible l'expérimentation de plusieurs solutions dans un temps raisonnable, avant de se décider sur une solution particulière et de procéder à son implémentation. Et d'autre part de l'efficacité des techniques de recherche qui permettent de trouver une solution au problème posé, sans parcourir pour autant tout l'espace des solutions, ce qui permettrait de raccourcir considérablement le temps de développement.

D'ici vient le besoin de pouvoir estimer de manière efficace les performances des choix d'implantation au niveau de la spécification algorithmique. Une méthode d'estimation qui soit assez rapide et précise et qui s'intègre facilement dans le flot de conception sera un bon candidat pour répondre à nos besoins. Pour ce faire, nous avons développé un modèle prédictif des performances globales, appelé modèle de caractérisation, indépendant du type de circuit, basé sur la composition des caractéristiques de chaque opérateur (surface, temps de réponse), caractéristiques obtenues par les outils de simulation appliqués à chaque opérateur isolé. Les différents composants sont caractérisés *à priori* en termes de surface et vitesse constituant ainsi une bibliothèque de caractéristiques de base pour le modèle.

Pour ce qui concerne les techniques d'exploration, comme notre cadre d'étude est le cadre des systèmes temps réel, nous nous intéressons à avoir une solution d'implantation qui respecte la contrainte temporelle temps réel du système que l'on étudie. En effet, nous avons vu dans l'introduction que la notion de temps réel n'est pas une notion portant sur la minimisation de la durée d'exécution du système mais plutôt une notion de respect de délai du système qui interagit avec son environnement. Ainsi notre problème de recherche se restreint à une solution valide sous-optimale dite optimisée mais pas nécessairement optimale. L'utilisation des méthodes approchées ou heuristiques qui restreignent l'espace à explorer en évitant les pistes sans intérêt tout en garantissant la qualité de la solution finale, s'avère répondre parfaitement à nos besoins. Contrairement aux méthodes exactes, ces méthodes

présentent l'intérêt de fournir rapidement une solution approchée au problème posé "solution sous-optimale acceptable" en un temps raisonnable sans avoir à explorer exhaustivement tout l'espace de solutions. C'est pourquoi, nous avons développé deux algorithmes d'optimisation de type heuristiques l'un glouton et l'autre basé sur le recuit simulé.

Dans la suite de ce chapitre, nous discutons notre méthode de caractérisation matérielle, en termes de surface et de latence, et nous décrivons notre méthode d'optimisation de l'implantation par défactorisation, et proposons des heuristiques.

6.2 Modèle d'estimation des ressources et des performances : caractérisation

Comme l'objectif recherché est de pouvoir évaluer et comparer l'ensembles des implantations valides en vue de déterminer l'implantation optimisée qui respecte les contraintes temporelles, une comparaison automatique des performances (latence, cadence, quantités de ressources requises) des différentes implantations possibles est donc indispensable et surtout elle doit se faire sans que l'utilisateur n'ait à les réaliser.

En effet, réaliser chaque implantation possible pour mesurer ses performances est inabordable : la durée de la réalisation d'une implantation peut être longue et il se peut que l'architecture ne soit pas encore disponible au moment où l'on veut faire la comparaison. La comparaison systématique par la mesure étant lente et très coûteuse elle est jugée inapplicable. Il est donc nécessaire de construire un modèle prédictif de performances (temps, surface) basé sur l'estimation qui soit suffisamment précis et fiable afin de guider de manière efficace l'exploration de l'espace des solutions par l'heuristique d'optimisation d'implantation. Les valeurs fournies par ce modèle d'estimation pendant la phase d'exploration doivent renseigner le concepteur sur la qualité des solutions envisagées, afin de prédire les résultats de l'implantation matérielle finale, sans pour autant qu'il soit obligé d'aller jusqu'à l'exécution de l'application sur l'architecture-cible. Il est clair que cette deuxième façon de procéder présente des sacrifices par rapport à l'exactitude des mesures. On évalue par estimation et donc approximativement la performance d'une implantation éventuelle sans la réaliser.

La qualité de cette estimation des ressources matérielles nécessaires à l'implantation matérielle et de ses performances temporelles dépend du compromis entre la précision envisagée et le temps de calcul consacré à ces estimations. Madsen et al. [111] recommandent que, pendant la phase d'exploration de l'espace des solutions, l'estimation soit plus rapide que dans la phase d'implantation finale, par conséquent, moins précise. Ainsi, les méthodes d'estimation plus précises et plus lentes sont utilisées lors de l'implantation finale, après avoir choisi les transformations à appliquer sur la spécification. Évidemment, les mesures les plus précises sont obtenues après la synthèse de l'implantation et son exécution sur l'architecture-cible. Nous nous intéressons dans le cadre de ce travail donc à une estimation rapide de l'architecture à un niveau algorithmique de la spécification.

Généralement, les techniques d'estimation de consommation de ressources matérielles et de temps de réponse ne sont pas très précises, puisque la mise en correspondance entre la description comportementale et l'implantation matérielle n'est pas directe (un-pour-un) [109]. Les compilateurs et les outils de placement-routage effectuent des optimisations inconnues par l'estimateur. Ainsi, il est très difficile de prédire les réductions de surface ou de temps de réponse engendrées par ces outils. Nous nous

contentons donc d'une prédiction qui, tout en n'étant pas grossière, est capable de guider les décisions des heuristiques d'optimisation.

Dès lors le modèle prédictif d'estimation de performances globales que nous proposons est basé sur une composition des caractéristiques des opérateurs (surface, temps de réponse), obtenues par les outils de simulation appliqués à chaque opérateur isolé. En fait, les différents types de composants obtenus après traduction matérielle doivent être caractérisés *a priori* en termes de surface et latence, constituant ainsi une bibliothèque de base pour notre modèle d'estimation. Ces composants élémentaires varient en fonction de l'application-cible, pouvant ainsi être des additionneurs, des soustracteurs, des multiplicateurs, des comparateurs, etc. Cette caractérisation matérielle préalable est nécessaire pour l'étiquetage des différents sommets du graphe algorithmique avec les valeurs de surface et de latence correspondant aux opérateurs qui les implantent. Cet étiquetage sera donc essentiel pendant l'étape d'estimation de surface et de latence de l'implantation de la spécification comme nous allons l'expliciter par la suite.

6.2.1 Caractérisation en surface

L'estimation de la surface occupée par l'implantation d'un algorithme sur une architecture circuit est généralement difficile à évaluer, étant donné la surface perdue lors du placement routage qui est délicate à prédire. En effet, le placement et le routage d'une architecture sont confiés à un logiciel externe, sur lequel nous avons peu de contrôle. Typiquement un logiciel tel que "**Alliance**" peut mettre plusieurs heures pour réaliser le routage de certains algorithmes. De plus les surfaces perdues varient d'un programme à un autre, ce qui nous rend difficile d'utiliser des méthodes génériques. Pour prendre en compte cette surface perdue, on procédera donc de façon empirique en incorporant une variable multiplicative α qui servira à moduler les résultats. Cette variable sera déduite d'un nombre d'essais le plus important possible pour coller le plus possible à la réalité. Ainsi, lors de l'estimation de la surface totale ST d'une architecture, on se contente donc d'additionner à un coefficient près α (dû au placement et routage) la totalité de la surface occupée par le chemin de donnée SD "*data-path*" et celle occupée par la logique de contrôle associée SC "*control-path*" : $ST \leftarrow \alpha \cdot (SD + SC)$.

Estimation de la surface du chemin de données

L'estimation de la surface occupée par le chemin de données se résume simplement à l'addition de la totalité des surfaces des différents sommets du graphe algorithmique étiqueté. Cependant du point de vue de la mise en oeuvre l'approche est légèrement différente. En effet, comme les opérateurs du graphe d'implantation matériel G_{im} issus de la traduction du graphe d'algorithme G_{al} resteront regroupés en frontière, on considère comme élément de base pour l'estimation la surface des frontières au lieu de celle des sommets. L'estimation se résume ainsi en la somme des surfaces des frontières constituants le graphe matériel d'implantation G_{im}, de cette manière le calcul est récursif et le traitement se fait avec une complexité linéaire, chaque sommet n'étant traité qu'une seule fois.

Étant donné que la surface du *data-path* d'une frontière FF donnée contient :

- la surface des opérateurs de factorisation situés sur la frontière ; (voir figure 6.1)
- la surface des opérateurs directement inclus dans la frontière, un sommet étant directement inclu s'il est contenu dans la frontière, mais pas dans les sous-frontières (cas d'imbrication de frontières) ;(Voir figure 6.2)

– la surface occupée par les sous-frontières (frontières incluses dans la frontière).(Voir figure 6.3).

La surface du chemin de donnée *SD " data-path"* du graphe matériel se calcule alors en appliquant récursivement l'algorithme 4 page 115 à partir de la frontière infinie globale englobant l'algorithme de l'application.

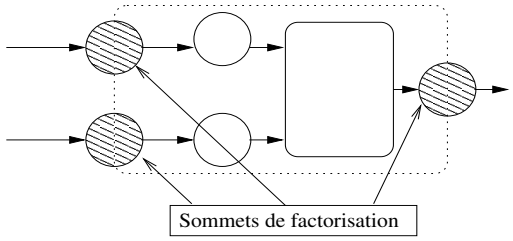


FIG. 6.1 – Surface des opérateurs de factorisation d'une frontière

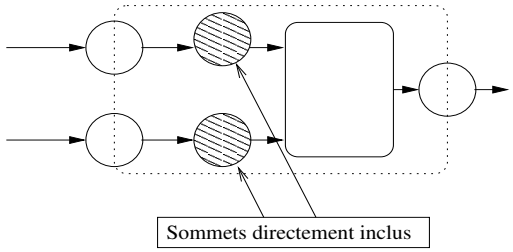


FIG. 6.2 – Surface des opérateurs directement inclus dans une frontière

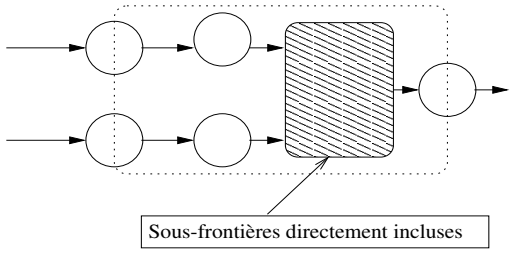


FIG. 6.3 – Surface des sous-frontières incluses dans une frontière

Algorithm 4 Surface du *data-path*

Entrée : Graphe algorithmique étiqueté par les surfaces de ses sommets : $G_{al} = (O, D)$.
Sortie : La surface $SD_{data}(FF)$ occupée par le *data-path* de la frontière FF.
Begin
Soit OF l'ensemble des opérateurs de factorisation situés sur la frontière FF.
Soit IF l'ensemble des sommets directement inclus dans la frontière FF.
Soit SF l'ensemble des sous-frontières incluses dans la frontière FF.
1. $SD_{OF}(FF) \leftarrow \sum_{o \in OF} S(o)$ {surface de factorisation}
2. $SD_{IF}(FF) \leftarrow \sum_{o \in IF} S(o)$ {surface de calcul}
3. $SD_{data}(FF) \leftarrow SD_{OF}(f) + SD_{IF}(f) + \sum_{f' \in F} SD_{data}(f')$
End

Estimation de la surface du chemin de contrôle

L'estimation de la surface occupée par le *control-path* se décompose en l'évaluation de la surface occupée par les unités de contrôle proprement dites et en la surface des fonctions logiques "ET" nécessaires à la jonction de signaux de requête et d'acquittement de frontières en parallèles et des multiplexeurs "MUX" nécessaires à la sélection entre les signaux de requête des frontières conditionnées. L'algorithme de calcul se résume ainsi à l'algorithme 5 qui réalise ce calcul.

Algorithm 5 Surface du*control-path*

Entrée : Graphe d'interconnexions des unités de contrôle $G_{cc} = (O"_2, D"_2)$
Sortie : La surface $S_{control}$ occupée par le *control-path*
begin
$S_{control} \leftarrow \sum_{o"_{UC} \in O"_2} S_{UC}(o"_{UC}) + \sum_{o"_{ET/OU} \in O"_2} S_{ET/OU}(o"_{ET/OU}) + \sum_{o"_{Mux} \in O"_2} S_{Mux}(o"_{Mux})$
end

6.2.2 Caractérisation temporelle

Par définition la latence d'une implantation correspond à la durée totale d'exécution de l'algorithme implanté, c'est-à-dire le temps qu'il faut pour que toutes les données disponibles en entrée (capteur), soient traitées par l'algorithme et que le résultat soit disponible en sortie (actionneur). C'est donc la durée totale d'une seule itération du graphe d'implantation (i.e temps de réponse de l'implantation donnée). Cette latence L est égale au nombre de cycles N_{Cycles} multiplié par le temps d'horloge T_{Clock} : $L = N_{Cycles} \times T_{Clock}$. L'estimation de la performance temporelle d'une implantation d'un GFCDD en termes de latence exige donc l'évaluation de ces deux paramètres.

Temps d'horloge : graphe temporel

Le temps d'horloge T_{Clock} se définit par la durée nécessaire pour que tous les signaux du circuit soient électriquement stables après le front montant de l'horloge. Pour calculer ce temps d'horloge, on cherche à évaluer le chemin le plus "long" (appelé **chemin critique**) que les données empruntent dans le circuit correspondant à l'implantation matérielle. C'est le chemin le plus long qui sépare généralement : deux registres consécutifs, une entrée du circuit et un registre ou un registre et une sortie du circuit. La

longueur de ce chemin critique T_{cc}, une fois identifié, détermine la valeur minimale du temps d'horloge globale du circuit T_H : $T_H >= T_{cc}$ et offre ainsi au concepteur une prédiction de la fréquence maximale de l'architecture d'implantation à synthétiser.

Pour identifier l'ensemble de ces différents chemins dans le circuit d'implantation dont le graphe d'algorithme est la représentation de haut niveau, nous utilisons **un graphe temporel** noté G_T, que nous construisons par transformations successives à partir du graphe d'algorithme initial. L'intérêt principal de ce graphe temporel est la mise en évidence des sommets opérateurs qui présentent une stabilité électrique (ceux contenants des registres par exemple) tout en offrant une vue temporelle du circuit. Notons cependant que la notion de stabilité électrique n'est pas liée directement à la présence de registres, elle permet juste de préciser une coupure ou "scission" du chemin critique global. Une telle coupure du chemin peut avoir plusieurs origines. Elle peut bien évidemment être due à la présence d'un registre qui coupe le flot de données, mais on peut considérer qu'il y a aussi une rupture du chemin critique en un point dont on est sûr qu'il est dans le même état que le cycle précédent, une constante par exemple coupe le chemin critique et de ce fait constitue un point de stabilité, sans pour autant être constituée de registre. Ainsi du point de vue structurel, le graphe temporel permet de stocker l'ensemble des chemins critiques existant au sein du circuit d'implantation en mettant en évidence ces différents points de stabilité. Cependant, du point de vue temporel il permet de fournir tous les paramètres temporels du circuit ce qui le rend utile pour le calcul du temps de cycle comme nous allons l'illustrer dans ce qui suit.

Construction du graphe temporel Le graphe temporel GT s'obtient alors, par transformations successives du graphe d'algorithme G_{al}. Pour cela on remplace d'abord tous les sommets purement combinatoires par un sommet pondéré par le temps de retard associé (*temps de traversée*). Puis on marque tous les points de stabilité électrique par un nouveau sommet noté $STAB$. Ces points de stabilité peuvent être au niveau d'un arc, dans ce cas on divise l'arc en deux et on intercale ce sommet, ou au niveau de l'intérieur même d'un sommet de type séquentiel (exemple des registres) dans ce cas on intercale ce sommet entre deux autres sommets traduisant le temps de retard en entrée et en sortie. On recherche ensuite sur le graphe GT ainsi construit le chemin le plus long, "chemin critique", permettant le calcul de la valeur de la période d'horloge.

Problèmes Il est facile de connaître l'état de stabilité électrique d'un certain nombre de points du GFCDD, notamment les opérateurs de factorisation infinis ($Fork^{\infty}$ "capteurs", $Join^{\infty}$ "actionneurs", $Itérate^{\infty}$ "retard" et $Diffuse^{\infty}$ "constante") qui à tous les cycles d'horloge fournissent des données stables grâce aux registres qu'ils contiennent. Il en va autrement des opérateurs de factorisation finie dont l'état de stabilité des informations entrantes et sortantes n'est pas toujours facile à connaître puisqu'il varie en fonction du type de cycle de factorisation pendant lequel se déroule la propagation des données. En effet, contrairement aux opérateurs traditionnels, ces opérateurs n'ont pas un comportement identique à chacun des cycles de factorisation : on est donc obligé de les considérer à différents moments. Dès lors pour une frontière factorisée par un facteur f, nous distinguons trois types de cycles différents correspondant chacun à un état de stabilité électrique différent :

1. cycle **initial**, premier cycle de la factorisation ;
2. cycles **intermédiaires**, cycles 2 à $f-1$ si $f > 2$;

3. cycle **final**, dernier cycle de la factorisation.

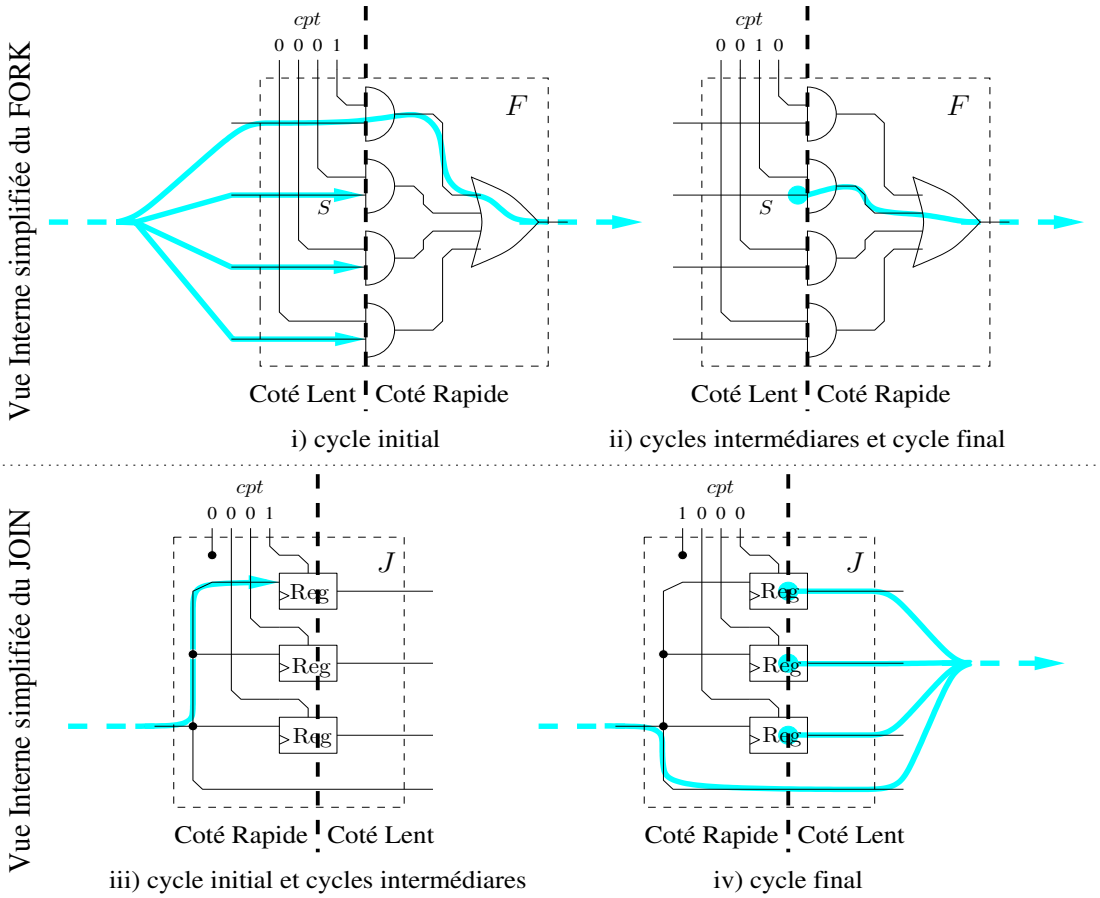


FIG. 6.4 – Propagation des signaux dans les opérateurs d'entrée et de sortie d'une frontière de factorisation

Lors du cycle initial les opérateurs d'entrée d'une frontière, comme le $FORK$, laissent passer directement l'information venant des frontières productrices amont. De même les opérateurs d'entrée d'une frontière, comme le $JOIN$ laissent passer directement la dernière donnée calculée, pendant le cycle final, vers les frontières aval, comme le montre les figures 6.4 où le facteur de factorisation $f = 4$.

Ainsi, comme le montre la figure 6.5, on peut considérer que la propagation des signaux électriques commence dans la logique amont des opérateurs d'entrée au cycle initial, se termine dans la logique aval des opérateurs de sortie au cycle final, alors qu'à tous les autres moments la propagation commence, au niveau des entrées des opérateurs d'entrée.

Pour résoudre ce problème nous avons été contraint de construire trois graphes temporels par frontière, décrivant chacun un type de cycle différent, que nous fusionnons par la suite en ne gardant qu'un seul sommet stabilité $STAB$ dans le graphe temporel fusionné de la frontière. Nous obtenons ainsi un unique graphe temporel GT_f par frontière regroupant les informations relatives aux trois cycles (i.e, les trois graphes temporels : $GT_{f_{init}}, GT_{f_{inter}}, GT_{f_{final}}$) de la frontière de factorisation en question. Enfin, après cette transformation, nous pouvons réduire ce graphe temporel G_T à l'aide d'algorithmes

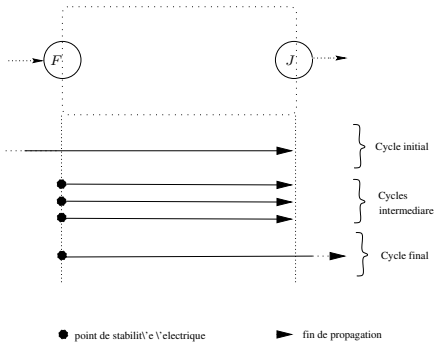
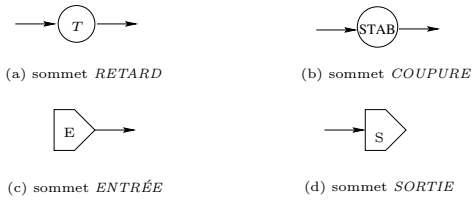


FIG. 6.5 – Propagation des signaux



(a) sommet *RETARD* (b) sommet *COUPURE*

(c) sommet *ENTRÉE* (d) sommet *SORTIE*

FIG. 6.6 – Les sommets de base du graphe temporel

de recherche de chemin maximal en vue de ne garder que les informations temporelles relatives aux chemins les plus longs entre les entrées et le sommet unique $STAB$, entre le sommet $STAB$ et lui même, et entre le sommet $STAB$ et les sorties.

Ce processus de construction est appliqué itérativement à chaque frontière de factorisation en commençant d'abord par les frontières n'incluant aucune autre, et on finissant par la frontière racine. Pour mener à terme le processus de construction du GT_f nous avons été obligés d'introduire deux nouveaux types de sommets permettant cette construction hiérarchique inverse :

- le sommet *ENTRÉE* qui ne peut avoir que des arcs sortants et qui symbolisent l'entrée de la frontière de factorisation finie,
- le sommet *SORTIE* qui ne peut avoir que des arcs sortants et qui symbolisent la sortie de la frontière de factorisation finie.

Techniquement, ces sommets sont associés à des points de référence du graphe non stables au cours du temps. Pour évaluer la longueur d'un chemin passant par un de ces points, il faudra prendre en compte la logique amont (pour le sommets *ENTRÉE*) ou aval (pour les sommets $SORTIE$). Ces sommets nous permettent de couper temporairement le graphe temporel global GT en cours de construction (graphe temporel de l'implantation), pour traiter les sous-graphes temporels obtenus séparément ($GT_{f_{init}}, GT_{f_{inter}}, GT_{f_{final}}$). De ce fait, un graphe temporel est formé de quatre types de sommets : $STAB$, ENTREE(E), SORTIE(S) ainsi que le sommet temporel T identifiant la latence (retard) du sommet opérateur correspondant (voir figure 6.6).

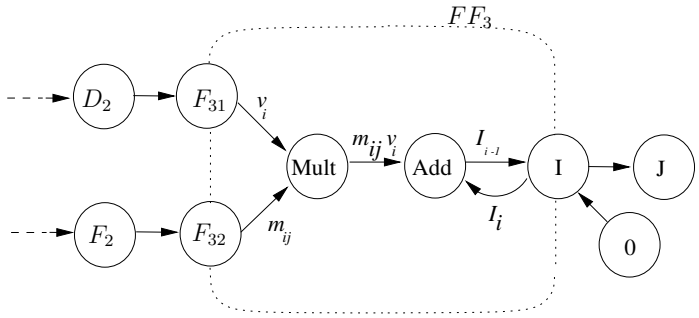


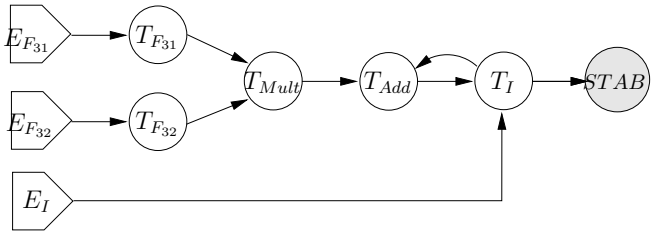
FIG. 6.7 – Sous graphe matériel correspondant à FF_3

Exemple de construction d'un GT_f Concrètement, nous allons maintenant expliquer comment créer le graphe temporel GT_f d'une frontière f. À chaque étape, une explication générale de la méthode sera présentée, puis cette dernière sera directement appliquée à un exemple : la frontière FF_3 de la figure Fig. 4.13 factorisant le calcul du produit scalaire représentée dans la figure 6.7.

1. Création des 3 graphes

- Le graphe temporel $GT_{f_{init}}$

Pour générer le graphe temporel $GT_{f_{init}}$ associé au **cycle initial** de la frontière considérée f, on remplace chaque sommet par son temps de traversée, puis on remplace chaque arc pointant vers un sommet d'entrée de la frontière f (F_{31}, F_{32}, I pour FF_3 sur l'exemple) par un sommet ENTRÉE et chaque arc sortant d'un sommet de sortie de la frontière f par un sommet $STAB$ marquant un point de stabilité du flot de données (le sommet I sur l'exemple)(voir figure 6.8).

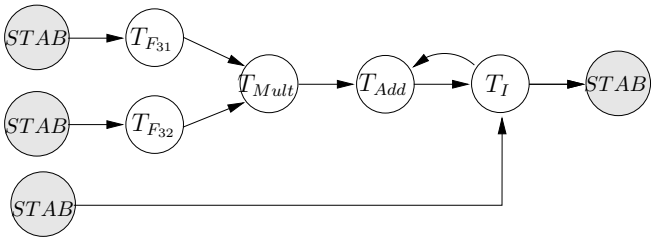


GT au cycle 1

FIG. 6.8 – Graphe temporel associé au Cycle Initial

- Le graphe temporel $GT_{f_{inter}}$

Pour générer le graphe temporel $GT_{f_{inter}}$ associé aux **cycles intermédiaires** de la frontière f considérée, on remplace chaque sommet par son temps de traversée, puis on remplace tous les arcs pointant vers un sommet d'entrée de la frontière f et chaque arc sortant d'un sommet de sortie de la frontière f par un sommet $STAB$ marquant un point de stabilité du flot de données (voir figure 6.9).

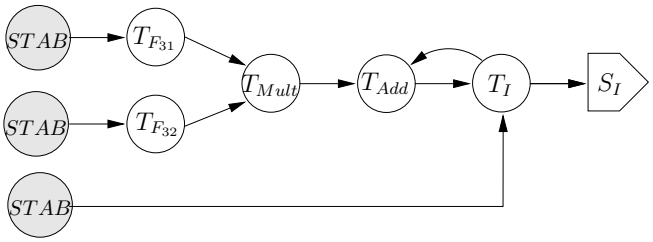


GT aux cycles 2 à f-1

FIG. 6.9 – Graphe temporel associé aux Cycles Intermédiares

- Le graphe temporel $GT_{f_{final}}$

Pour le graphe temporel $GT_{f_{final}}$ associé au **cycle final** on procède de même, en remplaçant les arcs sortant d'un sommet de sortie par un sommet SORTIE (voir figure 6.10).



GT au cycle f

FIG. 6.10 – Graphe temporel associé au Cycle Final

2. Fusion des 3 graphes

La deuxième étape consiste à fusionner tous les sommets $STAB$ des trois graphes formés précédemment en un et un seul sommet de stabilité $STAB$. On obtient ainsi un unique graphe temporel regroupant les informations des trois cycles :

3. Déduction du graphe temporel GT_f

Enfin, pour extraire tous les temps caractéristiques de la frontière f que l'on traite, afin de les mettre en évidence sur le graphe temporel réduit GT_f, il suffit de rechercher :

- les chemins de longueur maximal entre chaque sommet ENTRÉE et le sommet $STAB$; on obtient ainsi le **temps d'entrée** associé à chaque sommet d'entrée de la frontière (T_{E_I} *pour le sommet* E_I,- $T_{E_{F_{31}}}$ *pour le sommet* $E_{F_{31}}$, $T_{E_{F_{32}}}$ *pour le sommet* $E_{F_{32}}$ sur l'exemple traité). Notons que comme chaque sommet d'entrée ($E_I, E_{F_{31}}, E_{F_{32}}$) dispose d'un chemin unique le reliant au sommet $STAB$,

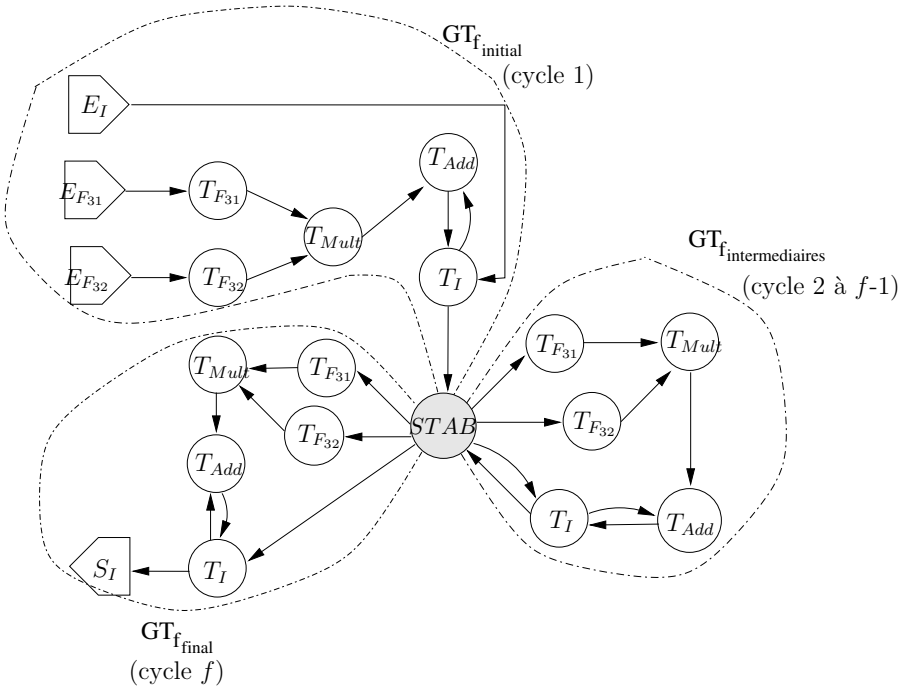


FIG. 6.11 – Graphe temporel Fusionné

le chemin maximal recherché pour chacun de ces sommets d'entrée est identique dans chaque cas au chemin unique identifié. Les longueurs des chemins respectifs sont :
$T_{E_I} = T_{I_{in}} + T_{Add} + T_{I_{out}}$, $T_{E_{F_{31}}} = T_{F_{31}} + T_{Mult} + T_{Add} + T_I$,
$T_{E_{F_{32}}} = T_{F_{32}} + T_{Mult} + T_{Add} + T_I$;

– le chemin de longueur maximal entre le sommet $STAB$ et lui-même (i.e. cycle) ; on caractérise ainsi le **temps interne** de la frontière ; sur l'exemple qu'on traite ce temps est égal à la valeur maximale des longueurs des chemins suivants :
$T_{Min} = \text{Max}[(T_{F_{31}} + T_{Mult} + T_{Add} + T_I),$
$(T_{F_{32}} + T_{Mult} + T_{Add} + T_I),$
$(T_I + T_{Add} + T_I)]$.

– les chemins de longueur maximale entre le sommet $STAB$ et chaque sommet SORTIE ; on obtient ainsi le **temps de sortie** associé à chaque sommet de sortie de la frontière ; sur l'exemple qu'on traite ce temps noté T_S est égal à la valeur maximale des longueurs des chemins suivants :
$T_{out} = \text{Max}[(T_{F_{31}} + T_{Mult} + T_{Add}),$
$(T_{F_{31}} + T_{Mult} + T_{Add}),$
$(T_I + T_{Add})]$.

De façon identique, on utilisera ensuite ces paramètres pour évaluer les propres temps caractéristiques de la frontière incluante jusqu'à parvenir à la dernière frontière infinie modélisant l'algorithme complet.

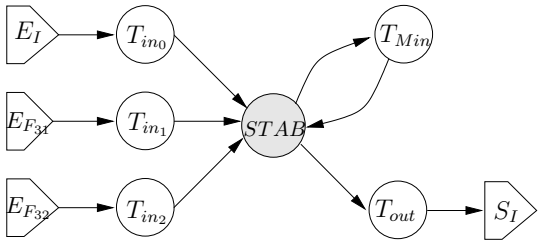


FIG. 6.12 – Graphe temporel GT_f de la frontière

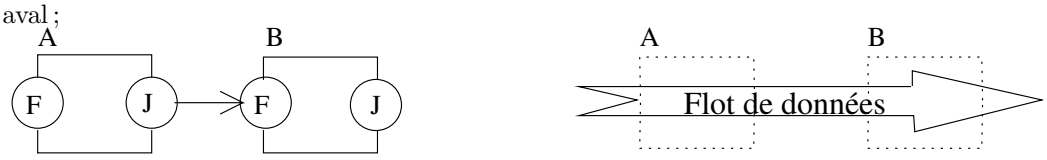
La longueur maximale à ce niveau est la longueur du chemin critique du circuit correspondant dans son intégralité. C'est ce dernier paramètre $T_{critique}$ qui détermine le temps de cycle d'horloge : $T_H >= T_{critique}$. Pour des raisons de longueur, la description complète de l'exemple n'apparaît pas ici, elle fait l'objet d'une présentation détaillée dans l'annexe A en fin de ce manuscrit.

Pour conclure, ce processus décrit à travers l'exemple peut être formulé par l'algorithme récursif 6 qui doit être appliqué à chaque fois qu'une frontière est rencontrée lors de la construction du graphe temporel global GT.

Calcul du nombre de cycles

Comme dit précédemment, l'estimation du temps d'exécution de l'implantation (i.e. latence) ne se limite pas seulement à la connaissance du temps d'horloge mais il faut aussi connaître le nombre total de cycles d'horloge nécessaires à l'exécution de l'implantation correspondant à l'algorithme spécifié. Pour calculer ce nombre, il suffit d'étudier les relations de voisinage entre les différentes frontières du graphe, représentées explicitement au niveau du graphe de voisinage G_v. Nous distinguons, en effet, trois grands types de relations de voisinage de frontières, auxquels correspondent des opérations de calculs spécifiques sur les nombres de cycles :

- relation en série (type +) : le nombre de cycles d'une configuration en série, où une frontière est productrice et l'autre est consommatrice. Il s'obtient en additionnant simplement le nombre de cycles de chacune car le flot de données traverse successivement et indépendamment chacune des frontières. Notons qu'on retranche 1 au nombre de cycles, car le dernier cycle de la frontière en amont utilisé pour sortir de cette frontière est en fait également le premier cycle de la frontière en aval ;



$$Cycles_{total} = Cycles(A) + Cycles(B) - 1$$

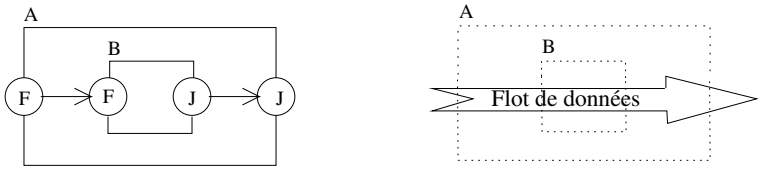
- relation d'inclusion (type x) : le nombre de cycles d'une telle configuration, où une frontière est imbriquée (incluse) dans une autre, s'obtient en multipliant simplement le nombre de cycles de chacune car à chaque fois que la frontière englobante exécute un cycle, la frontière englobée doit effectuer tous ses cycles de répétitions ;

Algorithm 6 Construction du Graphe Temporel d'une frontière

Entrée : Le graphe d'algorithme étiqueté $G_{al} = (O, D)$, la frontière f.
Sortie : Le graphe temporel réduit $GT_f = (O_{T_f}, D_{T_f})$ de la frontière f.
begin

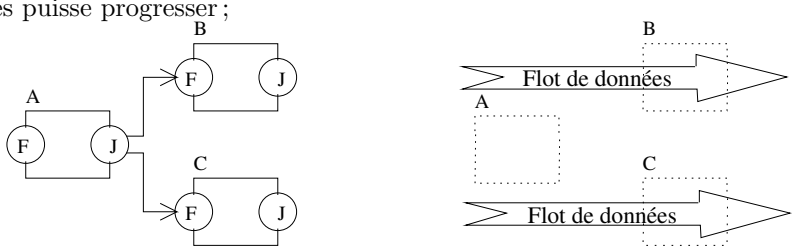
1: Créer un graphe temporel correspondant à chaque type de cycle en utilisant les règles (i.e les $GT_{f_{init}}$, $GT_{f_{inter}}$,$GT_{f_{final}}$) :
   - remplacer les sous-frontières incluses dans f par leur propre graphe temporel réduit construit récursivement ;
   - remplacer les sommets directement inclus dans f par leur équivalent temporel (sommet étiqueté par sa latence) ;
   - remplacer les sommets d'entrée et de sorties situés sur f par leur équivalent temporel correspondant au cycle considéré ;
     - Dans $GT_{f_{init}}$, faire précéder chaque sommet d'entrée de la frontière f par un sommet $ENTREE$, et faire suivre chaque sommet de sortie par un sommet $STAB$
     - Dans $GT_{f_{inter}}$, faire précéder chaque sommet d'entrée de la frontière f par un sommet $STAB$, et faire suivre chaque sommet de sortie par un sommet $STAB$
     - Dans $GT_{f_{final}}$, faire précéder chaque sommet d'entrée de la frontière f par un sommet $STAB$, et faire suivre chaque sommet de sortie par un sommet $SORTIE$
2: Fusionner les trois graphes obtenus, en remplaçant tous les sommets $STAB$ par un sommet unique $STAB$
3: Calculer les chemins de longueur maximale en considérant que le sommet de stabilité $STAB$ coupe tout chemin :
   - entre chaque sommet $ENTREE$ et le sommet $STAB$;
   - entre le sommet $STAB$ et lui même ;
   - entre chaque sommet $SORTIE$ et le sommet $STAB$;
4: Réduire le graphe en remplaçant chaque chemin maximal identifié entre les sommets $ENTREE$ et le sommet $STAB$ et entre les sommets $SORTIE$ et le sommet $STAB$, par un chemin ayant un seul sommet étiqueté par la longueur maximal calculée.

end



$$Cycles_{total} = Cycles(A) * Cycles(B)$$

– relation en parallèle (type Max) : dans une configuration en parallèle, les frontières n'ont pas de dépendances de données directes entre elles, mais elles sont en dépendance par l'intermédiaire d'une autre frontière. Ainsi le nombre de cycles total est égal au maximum (Max) des nombres de cycles des frontières parallèles car il est nécessaire que chaque traitement soit terminé pour que le flot de données puisse progresser ;



$$Cycles_{total} = Max(Cycles(A), Cycles(B))$$

Algorithm 7 Calcul du nombre de cycles

Entrée : Le graphe de voisinage $G_V = (O", D")$.
Sortie : Le nombre de cycles $Cycles_{total}$.
begin
Parcourir le graphe de voisinage $G_V = (O", D")$ et identifier les relations de voisinage entre ses sommets frontières.
Pour tous sommets frontières A et B de O" :
Si série(A,B) alors $Cycles_{total} = Cycles_{total} + [Cycles(A) + Cycles(B) - 1]$
Si inclus(A,B) alors $Cycles_{total} = Cycles_{total} + [Cycles(A) \times Cycles(B)]$
Si parallèle(A,B) alors $Cycles_{total} = Cycles_{total} + Max[Cycles(A), Cycles(B)]$
end

6.3 Optimisation de la latence

L'implantation de systèmes réactifs temps réel nécessite la considération de plusieurs critères souvent interdépendants et contradictoires (contraintes temps réel, surface et consommation..). La majorité de la littérature limite ainsi les méthodes proposées à un de ces critères du fait que l'amélioration de l'un des critères entraîne dans la plupart des cas une dégradation des autres.

Le critère que nous considérons principalement ici est le respect des contraintes temps réel, en particulier le temps de réponse (i.e. la latence). Dans nos applications cibles, ce critère est très important étant donné les conséquences dramatiques que peuvent avoir le non respect de ces contraintes (une centrale nucléaire ou un avion par exemple).

Nous nous intéressons donc en priorité au respect des contraintes temps réel en particulier la latence, tout en minimisant autant que possible la consommation des ressources en termes de surface. Plus précisément on cherche à avoir une durée d'exécution de l'algorithme d'application (i.e. latence) qui soit inférieure ou égale à la contrainte temps réel de latence du système que l'on étudie.

6.3.1 Défactorisation et parallélisme

Un des moyens de respecter les contraintes temps réel d'une application (en particulier sa latence) est d'augmenter sa puissance de calcul en utilisant une implantation parallèle ; il est donc clair qu'une mise en évidence de ce parallélisme au niveau de la spécification de l'algorithme de l'application facilite son exploitation quant cela est nécessaire lors de l'implantation.

Lorsqu'un utilisateur spécifie son algorithme sous la forme d'un graphe factorisé et conditionné de dépendances de données, il est plus simple pour lui de factoriser au maximum les motifs répétitifs sans avoir à se soucier du temps d'exécution de l'algorithme ainsi spécifié car cela réduit la taille de la spécification.

Une implantation séquentielle répétitive directe de cette spécification factorisée entraîne une utilisation minimale de surface en dépit d'un temps de calcul élevé étant donné qu'elle n'exploite pas le parallélisme potentiel des calculs spécifiés dans les frontières de factorisation. Dès lors, si la latence de cette implantation directe ne respecte pas les contraintes temps réel de latence, il va falloir procéder à une transformation du graphe d'algorithme correspondant à la spécification initiale par défactorisation des frontières de factorisation afin de tirer profit du parallélisme des calculs et obtenir ainsi une implantation plus parallèle procurant de meilleures performances temps réel au prix de ressources supplémentaires (surface de circuit). Défactoriser une frontière consiste donc à remplacer cette frontière par plusieurs frontières représentant le même motif de répétition, mais dont la somme des répétitions est égale à la répétition de la frontière initiale. Ces nouvelles frontières pouvant ainsi s'exécuter en parallèle, le temps de calcul en est diminué, mais en contrepartie la surface de l'implantation est augmentée. La défactorisation étant la transformation inverse de la factorisation, elle ne change donc pas le parallélisme potentiel de l'algorithme spécifié, elle a pour intérêt de le mettre en évidence en vue de permettre une implantation plus parallèle.

À chaque défactorisation d'une spécification correspond une implantation plus ou moins parallèle. Le plus souvent, une implantation plus défactorisée nécessite plus de ressources matérielles mais moins de latence et/ou cadence qu'une implantation moins défactorisée, car les données traitées en parallèle par des opérateurs différents dans le cas plus défactorisé sont multiplexées (séquencées temporellement) pour être traitées par le même opérateur dans le cas moins défactorisé. Par exemple sur la Fig. 6.13 représentant l'implantation du C-PMV partiellement défactorisé par un facteur de 2, le calcul des différents produits scalaires sera fait en parallèle au niveau des deux sous-graphes (FF_{2_a}, FF_{3_a}), (FF_{2_b}, FF_{3_b}) ce qui permet de réduire la latence du circuit mais en contrepartie augmente la surface d'implantation : augmentation du nombre d'opérateurs frontières de factorisation et d'opérateurs de calcul, des unités de contrôle UC_1, UC_{2a}, UC_{2b}, UC_{3a} et UC_{3b} et ajout de nouveaux opérateurs X_1 et M_1 qui implantent la défactorisation partielle des données (X : étant un opérateur de décomposition de tableaux de données et M : l'opérateur de regroupement de données en tableau).

Parmi toutes les transformations possibles par défactorisation, on éliminera celles qui ne respectent pas les contraintes temps réel, et on cherchera celle qui minimise les ressources nécessaires à l'implantation tout en respectant les contraintes temporelles.

6.3.2 Recherche de l'optimum

Comme il a été mentionné précédemment, le problème de recherche d'implantation optimisée d'un algorithme factorisé sur une architecture circuit est formalisé en termes de transformations de graphes,

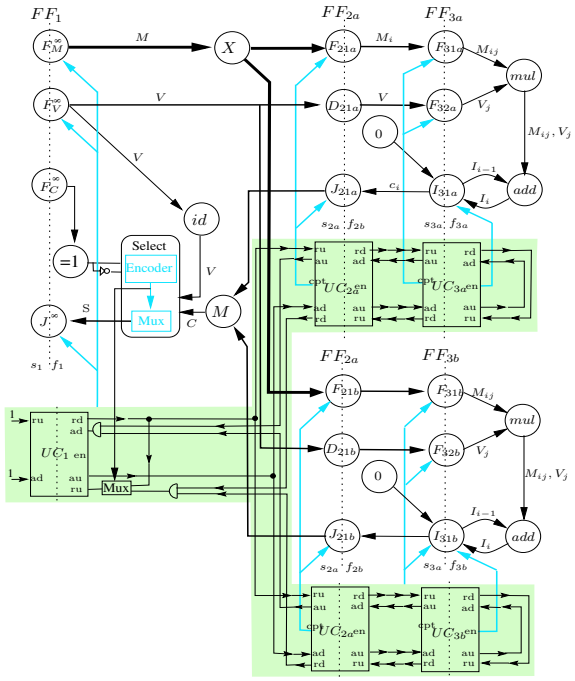


FIG. 6.13 – Implantation matérielle du C-PMV défactorisé partiellement

i.e. défactorisations. L'espace des solutions à explorer pour retrouver la meilleure solution est composé donc de l'ensemble de toutes les défactorisations possibles du graphe factorisé spécifiant l'algorithme.

Pour un graphe d'algorithme ayant n frontières, nous avons au minimum 2^n implantations défactorisées. De plus, chaque frontière peut n'être défactorisée que partiellement : une factorisation de r répétitions d'un motif peut se décomposer en f frontières de factorisation de r/f répétitions du motif. Par conséquent, pour un graphe d'algorithme donné, on a un nombre extrêmement grand, mais fini, de défactorisations possibles qui sont plus ou moins défactorisées, et parmi lesquelles nous avons à sélectionner l'état de défactorisation qui garantisse une surface d'implantation minimale tout en satisfaisant les contraintes temps réel.

Nous sommes face ainsi à un problème d'explosion combinatoire et la recherche de l'implantation optimisée est par conséquent un problème d'optimisation dont la taille est généralement grande pour des applications réelles à cause de la complexité combinatoire exponentielle de la recherche de la solution exacte (optimisée).

6.3.3 Méthodes de résolution

Pour résoudre ce type de problème, on distingue dans la littérature deux familles de méthodes de recherche permettant l'exploration de l'espace de solutions en vue de trouver l'implantation optimale : les méthodes exactes et les méthodes approchées ou heuristiques.

Les méthodes exactes Ce sont en général des méthodes énumératives, c'est-à-dire que les algorithmes associés à ces méthodes, reposent sur une exploration (et comparaison) de tout l'espace des solutions possibles. Ces méthodes trouvent toujours la solution exacte si on leur laisse suffisamment

du temps. Comme leurs temps d'exploration est très long en pratique, elles sont toutes limitées à des problèmes de petite taille. Elles sont très coûteuses et par conséquent totalement inadaptées aux applications temps réel qui sont en général des problèmes de grande taille. Parmi ces méthodes on distingue : les méthodes de programmation mathématiques, les méthodes séparation/évaluation (Branch and Bound), l'algorithme A*.

Les méthodes approchées ou heuristiques Ce sont des méthodes permettant de trouver une solution approchée au problème posé. L'espace des solutions n'est pas entièrement visité. Elles cherchent à trouver une solution dite sous-optimale c'est-à-dire la meilleure relativement au critère choisi par l'utilisateur. Elles ont l'intérêt de fournir rapidement une solution sous-optimale acceptable. Parmi ces méthodes on distingue : les approches dites gloutonnes et les méthodes de voisinage.

Heuristiques gloutonnes Ce sont des méthodes qui construisent itérativement une seule solution complète en prenant à chaque étape une décision (choix) qui semble la meilleure à cette étape (choix optimal localement) dans l'espoir que cette décision mènera à la solution optimale globale. Cette décision n'est jamais remise en cause par la suite, c'est une méthode incrémentale sans retour arrière "backtracking".

Heuristiques de voisinage Ces méthodes partent d'une solution initiale complète éventuellement calculée par une méthode gloutonne et elles cherchent à améliorer cette solution. Elles sont classées en deux catégories selon qu'elles reposent sur une méthode de recherche dans un voisinage local ou dans un voisinage global. Les méthodes recuit simulé, tabou, et algorithmes génétiques appartiennent à cette dernière catégorie de méthodes de recherche globale.

Comme nous nous intéressons à des applications temps réel dont la complexité ne cesse de croître et pour lesquelles nous recherchons l'implantation qui respecte les contraintes temps réel de latence, nous favorisons le recours aux méthodes heuristiques qui fournissent de bons résultats en un temps acceptable. Nous rappelons ici que la notion de temps réel n'est pas une notion portant sur la minimisation de la durée d'exécution (latence) mais plutôt une notion de respect de délai du système qui interagit avec son environnement. Ainsi une solution approchée sous-optimale qui n'est pas nécessairement l'optimale répond parfaitement à nos besoins. Les heuristiques sont guidées par une fonction de coût permettant la comparaison des performances des différentes défactorisations, nous permettant d'explorer seulement un sous-ensemble de toutes les défactorisations possibles et d'en trouver une implantation optimisée (pas nécessairement optimale), sans parcourir pour autant tout l'espace des solutions.

Puisque nous nous intéressons au prototypage rapide, nous avons privilégié dans un premier temps les "heuristiques gloutonnes" qui sont très rapides et donnent des résultats de bonne qualité.

6.3.4 Heuristique gloutonne

L'heuristique que nous proposons est un algorithme glouton qui construit une solution itérativement en prenant à chaque étape une décision concernant le degré de factorisation d'une frontière donnée. À chaque itération l'heuristique détermine une frontière à défactoriser et à chaque fois de combien la

défactoriser. Le but final étant de respecter la contrainte de temps pour une surface minimale. Pour cela, chaque itération se décompose en trois étapes :

1. déterminer quels sont les frontières intéressantes à défactoriser,
2. déterminer le facteur de défactorisation associé à chacune d'elles,
3. déterminer finalement la frontière qui sera défactorisée à la fin de l'itération.

Les frontières les plus intéressantes à défactoriser Rappelons que nous recherchons à réduire la latence du graphe pour qu'elle soit inférieure ou égale à la latence temps réel. L'optimisation est donc basée sur des calculs de chemins critiques sur le graphe de l'algorithme, étiqueté par les durées et la consommation des opérations déduites du modèle de caractérisation. L'heuristique s'applique à ne transformer que les frontières appartenant au chemin critique *cc* (séquence d'opérations la plus longue dans le circuit). En effet si l'on défactorise l'une de ces frontières le chemin critique sera "raccourci" et donc son temps d'exécution, i.e. la latence du circuit en sera diminuée. À chaque étape de l'heuristique on construit donc une liste des FF candidates à la transformation (FF appartenant au chemin critique) et pour chacune d'elles on détermine le facteur de défactorisation le plus intéressant.

Le facteur de défactorisation optimal d'une frontière Pour chacune des frontières précitées (i.e. traversée par le chemin critique) nous allons déterminer le facteur de défactorisation le plus intéressant, que nous qualifierons d'optimal. Ce facteur est :

- soit le plus grand facteur de défactorisation entraînant une diminution de la latence. Si ce facteur ne correspond pas au facteur de factorisation (cas d'une défactorisation totale), alors cela signifie que la frontière ainsi défactorisée n'est plus traversée par le chemin critique. De ce fait, pour les raisons précitées, il est inutile d'appliquer une défactorisation plus grande ;
  OU
- soit le plus petit facteur de factorisation impliquant une latence inférieure à la contrainte de temps. Il est inutile d'augmenter la surface d'un circuit lorsque sa latence satisfait la contrainte indiquée par l'utilisateur. Si ce facteur correspond au facteur de factorisation (cas d'une défactorisation totale), sans que la latence soit inférieure à la contrainte temporelle alors cela signifie que la frontière ainsi défactorisée n'est plus traversée par le chemin critique.

Choix de la frontière à défactoriser Une fois déterminés les facteurs de défactorisations optimaux pour les frontières précitées, il faut introduire une fonction de coût permettant de calculer pour chaque frontière sa "pression de défactorisation". Pour cela, il faut rappeler que l'on cherche à maximiser le gain en temps, minimiser les pertes en surfaces, tout en tenant compte de la contrainte de temps donnée par l'utilisateur. Nous proposons donc une fonction de coût qui permet de choisir parmi les frontières candidates à la défactorisation, celle qui engendrera une transformation optimale, c'est à dire celle qui génère le moins de surface pour une réduction maximale de la latence. Pour une frontière donnée cette fonction de coût sera déterminée comme suit :

$$f = \frac{\Delta S}{T - max\{T', C_t\}}$$

où ΔS représente la perte en surface, T la latence du circuit avant la défactorisation, T' la latence du circuit après défactorisation, et C_t la contrainte de temps. À la fin de l'itération, la frontière la plus

intéressante ainsi que son facteur de défactorisation sont déterminés. L'algorithme 8 décrit l'heuristique gloutonne proposée.

Algorithm 8 Algorithme Glouton d'optimisation

Entrée : Le graphe GFDD $G_{FCDD} = (O, D)$ de l'implantation, la contrainte temporelle C_t
Sortie : Le graphe défactorisé
Soit :
L : latence du graphe d'implantation G_{im} du G_{FCDD} avant défactorisation,
L' : latence du graphe d'implantation G_{im} du G_{FCDD} après défactorisation,
S : surface du graphe d'implantation G_{im} du G_{FCDD} avant défactorisation,
S' : surface du graphe d'implantation G_{im} du G_{FCDD} après défactorisation,

1: **begin**
2: **repeat**
3: Calculer la latence L du chemin critique CC du graphe d'implantation courant ;
4: Calculer la surface S courante de l'implantation courante ;
5: **while** $L > C_t$ **do**
6: Déterminer la liste des FF candidates à la défactorisation FF_{list}, i.e appartenant au chemin critique CC ; $FF_{List} := FF_i, FF_i \in CC$
7: **for all** frontière candidate $FF \in FF_{list}$ **do**
8: Déterminer son facteur de défactorisation optimale df_{FF} par l'algorithme 9 ;
9: Calculer la nouvelle surface S' correspondante à la défactorisation de FF par df_{FF} ;
10: Calculer la nouvelle latence L' correspondante à la défactorisation de FF par df_{FF} ;
11: Calculer le coût correspondant à cette défactorisation $f = S' - S/L - Max(L', Ctr)$
12: **end for**
13: Ordonner la liste des frontières FF_{list} par ordre de coût croissant ;
14: Défactoriser la frontière en tête de liste par son facteur de défactorisation df_{FF} ;
15: Le graphe ainsi défactorisé deviendera le graphe courant ;
16: **end while**
17: **end**

Algorithm 9 Algorithme de calcul du facteur de défactorisation d'une frontière

Entrée : la contrainte C_t, la frontière courante FF, le graphe G_{FCDD}
Sortie : facteur optimal de défactorisation df_{FF}

1: **begin**
2: df_{FF} := facteur_défactorisation_initial(FF)
3: $G'_{CFDD} := G_{CFDD}$
4: **while** (latence $(G'_{CFDD}) \geq C_t$) and ($df_{FF} \leq$ facteur_factorisation (FF)) **do**
5: $df_{FF} := df_{FF} + 1$;
6: $G'_{FDD} := G'_{FDD}$ avec FF défactorisée par df_{FF}
7: **end while**
8: **end**

6.3.5 Heuristique de voisinage : recuit simulé

Nous préconisons une heuristique de voisinage de type "recuit simulé" car elle dispose d'une propriété importante, qui est la preuve de sa convergence vers l'optimum global. Cependant, la critique principale faite à cette technique du recuit simulé est son extrême lenteur comparée avec des heuristiques gloutonnes pour des tailles de problèmes réalistes.

Principe du recuit simulé Cette technique s'inspire de la thermodynamique. En effet, les systèmes physiques atteignent leur état d'équilibre (minimum d'énergie) en dépit du nombre immense de configurations que peuvent prendre les particules constituant le système. Le principe physique de cette transition vers l'équilibre est : "un système qui est refroidi lentement, aura une structure très ordonnée et cette structure correspond à l'énergie minimale du système, inversement un système que l'on refroidit trop vite aura une structure peu ordonnée, autrement dit présentera des défauts".

La méthode du recuit simulé consiste à transposer ce procédé à la résolution d'un problème d'optimisation. Par analogie, le recuit simulé considère les solutions comme les configurations d'un système physique dont l'énergie est le coût que l'on cherche à minimiser. L'idée est de faire une simulation de ce système avec une probabilité d'acceptation d'une modification de la solution qui accroît l'énergie du système (le coût) définie selon une formule analogue à l'équation Boltzmann donnée par :

$$p = exp(-(E_2 - E_1)/kT)$$

où E_i sont les niveaux d'énergie, k est la constante de Boltzmann fixée à 1 pour le recuit simulé et T la température absolue. En faisant décroître le paramètre de température, au fil des itérations, le système se gèle dans une configuration de moindre énergie que la solution initiale. Le fait d'accepter avec une certaine probabilité le passage d'une solution à une solution moins bonne au lieu de la rejeter (augmentation de l'énergie) permet ainsi de ne pas rester bloqué dans un optimum local (et donc, en théorie, d'atteindre le minimum global). Les résultats obtenus avec ce type de calcul sont satisfaisants même si l'exécution est gourmande en temps de calcul.

Recuit simulé : algorithme de base La première étape du recuit simulé consiste à déterminer une solution initiale pour laquelle la température est fixée à une valeur élevée. Une solution est caractérisée par un grand nombre de variables. Les itérations du recuit simulé consistent ensuite à changer à chaque fois les valeurs de quelques variables dans la solution courante. L'ensemble des changements autorisés pour faire passer d'une solution à une autre est appelé "mouvement". Une itération consiste donc à choisir un mouvement qui, appliqué à la solution courante X_i, de coût $f(X_i)$, génère une solution "choisie" X_j, de coût $f(X_j)$.

- Si l'incrément de coût $\Delta f_{ij} = f(X_i) - f(X_j)$ est négatif, la solution choisie est meilleure que la solution courante et elle sera toujours acceptée comme nouvelle solution
- Si Δf_{ij} est positif, la solution choisie est acceptée comme nouvelle solution avec la probabilité $e^{\Delta f_{ij}/T}$. Lorsqu'elle n'est pas acceptée, la solution courante est conservée comme nouvelle solution. Pour une valeur de T, les itérations sont effectuées jusqu'à ce que le système soit en "équilibre thermique", c'est-à-dire que la moyenne du coût soit stabilisée, puis une nouvelle valeur de la température est sélectionnée et un autre ensemble d'itérations est effectué jusqu'à ce que l'équilibre soit atteint. Le processus se termine lorsque le système est gelé, c'est-à-dire que le coût n'est pas modifié au cours d'un grand nombre, fixé à l'avance, d'itérations consécutives. Cet algorithme est résumé dans l'algorithme 10 suivant :

Algorithme de recuit simulé proposé Dans notre cas d'étude, une solution est un vecteur d'état dont les variables décrivent l'état de factorisation/défactorisation des différentes frontières du GCFDD. Ces variables varient donc entre 1 et le facteur de répétition de la frontière respective. Le mouvement

Algorithm 10 L'algorithme de la méthode du recuit simulé

begin
Définir une solution initiale X_0.
Poser cette solution comme solution courante X_i.
Sélectionner une température initiale élevée.
while Le système n'est pas gelé **do**
  while Le système n'est pas en équilibre thermique **do**
    Choisir un mouvement, conduisant à une solution voisine X_j de coût $f(X_j)$;
    Poser $\Delta f_{ij} = f(X_i) - f(X_j)$;
    if $\Delta f_{ij} \leq 0$ **then**
      La solution choisie devient la solution courante ;
    else
      Calculer la probabilité d'acceptation $p = e^{\Delta f_{ij}/T}$
      Tirer au sort un nombre aléatoire a dans [0,1]
      if $a \leq p$ **then**
        La solution choisie devient la solution courante (acceptation)
      else
        Ne pas changer la solution courante ;
      end if
    end if
    Réduire la température T : $T \leftarrow kT$ $\;avec\; k < 1$
  end while
end while
Retourner la solution courante
end

d'une solution à une autre s'effectue en changeant aléatoirement la valeur du défacteur de l'une des frontières[1]. Ce changement d'état par défactorisation provoque une diminution ou une augmentation du temps d'exécution de l'algorithme implanté et de la surface utilisée. Pour pouvoir estimer la diminution/augmentation du temps d'exécution par rapport à l'augmentation/diminution de la surface, nous avons défini la fonction de coût suivante : soit
Ct la contrainte en temps,
t le temps d'exécution de l'algorithme, obtenu après une défactorisation,
k facteur de pénalité,
S la surface de l'implantation, obtenue après une défactorisation,
la fonction du coût pour un état X est $F(X)$

$$F(X) = \begin{cases} S & si \quad t < T \\ S + k \times (t - Ct) & si \quad t \geq T \end{cases}$$

6.3.6 Évaluation

Les résultats expérimentaux (chapitre 8) ont permis de montrer que l'heuristique de recuit simulé donne toujours de meilleurs résultats que l'heuristique gloutonne, ceci est dû à une recherche exhaustive de la meilleure solution en gardant toujours une trace de celle trouvée auparavant. Par contre elle est beaucoup plus lente pour trouver la solution par rapport à l'approche gloutonne ce qui peut la

[1] normalement en absence de critères spéciaux, le choix de la frontière s'effectue d'une manière aléatoire

Algorithm 11 L'algorithme de l'heuristique recuit simulé proposé

begin
Initialiser les paramètres : $N_{maxiter} = \cdots, iter = 1, \epsilon = \cdots, ...$
while $(iter \neq N_{maxiter})$ou$(T \neq \epsilon)$ **do**
  for all un nombre variant d'essais nb=1 à L_k faire **do**
    Tirer au sort une solution X_j appartenant au voisinage de X_i : $X_j = V(X_i)$;
    if $F(X_i) < F(X_j)$ **then**
      $X_i \leftarrow X_j$
    else
      if $e^{(-(F(X_j)-F(X_i))/T_k)} > Random.float(0,1))$ **then**
        $X_i \leftarrow X_j$;
      else
        Garder X_i ;
      end if
    end if
  end for
  $T_{k+1} \leftarrow T_k \times \alpha$;
  $L_{k+1} \leftarrow L_k \times \beta$;
  $iter++$
end while
end

rendre inadéquate au prototypage rapide. On favorise donc l'heuristique gloutonne dans une optique de prototypage rapide d'applications et une fois que l'utilisateur est satisfait de la solution d'implantation obtenue, il peut à ce moment l'améliorer en appliquant le recuit simulé à partir de la solution obtenue précédemment.

6.4 Conclusion

Dans ce chapitre nous avons présenté notre modèle d'estimation prédictif de performance au niveau comportemental des caractéristiques de l'implantation matérielle. Nous effectuons notre estimation à un niveau d'abstraction élevé, qui est le niveau comportemental, ce qui permet d'explorer l'espace de solutions plus facilement sans effectuer la synthèse RTL ou logique. Il faut tout de même souligner l'impossibilité de réaliser une estimation précise étant donné le niveau d'abstraction élevé auquel s'applique notre estimateur. Basées sur ces estimations de performances, les heuristiques développées explorent efficacement l'espace des solutions d'implantation possibles en vue de la recherche de l'implantation optimisée respectant les contraintes temps réel. La stratégie consistant à respecter la contrainte temporelle en appliquant un processus de défactorisation (transformation spatiale/temporelle équivalente à un déroulage de boucle) de la spécification afin de tirer profit du parallélisme des calculs et d'obtenir une implantation plus parallèle.

Chapitre 7

Génération du code VHDL

Dans les précédents chapitres nous avons défini comment, à partir d'un graphe d'algorithme, de caractéristiques architecturales et de contraintes, nous construisons le graphe d'implantation optimisée correspondant. Dans ce chapitre nous nous intéressons à la dernière transformation de notre flot dont l'objectif est de générer automatiquement, à partir du graphe d'implantation obtenue, le code RTL décrivant l'architecture circuit correspondante. L'approche préconisée est basée sur l'utilisation d'un macro code intermédiaire générique permettant de s'affranchir de la diversité et de l'évolution constante des langages de description matérielle des circuits.

7.1 Introduction

Lors du développement d'un système temps réel, certains concepteurs ne veulent consacrer qu'un minimum de leur temps à l'implantation de leur algorithme en cherchant dans un premier temps à valider rapidement le comportement et les fonctionnalités du système conçu. Ce premier type de conception vise le prototypage rapide, où les objectifs prioritaires consistent à démontrer rapidement la faisabilité de l'application et éventuellement un premier dimensionnement de l'architecture matérielle d'implantation pour estimer le coût d'industrialisation. D'autres concepteurs veulent tirer parti au maximum des performances de l'architecture. Ce deuxième type de conception vise l'industrialisation, où les objectifs prioritaires sont de réaliser une application (logicielle et matérielle) qui respecte toujours des contraintes temps réel, un certain degré de sûreté de fonctionnement, en respectant aussi les critères d'embarquabilité et souvent en essayant de coûter le moins cher possible. Ces deux derniers points conduisent naturellement à la minimisation de l'architecture.

Pour satisfaire les contraintes apparemment contradictoires de ces deux types de conceptions visant des objectifs différents, et ainsi simplifier les travaux de développement des applications temps réel en vue de réduire les cycles de développement, une mise en oeuvre d'un processus de génération automatique de code s'avère indispensable.

En effet, l'automatisation de la génération de code assure un passage direct des spécifications au code. Ceci permet de réduire considérablement le temps séparant la conception d'un système et son implantation effective, et donc les premiers tests sur le système réel. La génération automatique se prête donc particulièrement bien au besoin du premier type de conception qui vise le prototypage rapide des systèmes. Mais l'automatisation de la génération de code permet également d'obtenir des garanties sur le comportement et la qualité du code généré qu'il serait impossible d'avoir dans le cas classique d'un codage manuel. En effet, si le flot de développement est basé sur une approche formelle favorisant la recherche de l'implantation optimisée où chaque transformation a été validée, le code généré automatiquement sera correct par construction. De plus, il respectera bien les contraintes de performance, ce qui répond parfaitement aux besoins spécifiques du deuxième type de conception.

De ce fait, un processus de génération automatique de code se prête aussi bien à la génération de code de prototypage, où l'on cherche à valider le comportement et les fonctionnalités du système conçu, qu'à la génération de code de production nécessaire à l'industrialisation, où l'on cherche à produire du code taillé sur mesure respectant des contraintes de taille et de performances.

Partant de ces constats, nous avons intégré une étape de génération automatique de code dans notre flot d'extension d'AAA. Nous avons défini ainsi un ensemble de règles permettant de générer automatiquement et de manière systématique le code RTL décrivant l'architecture circuit supportant l'implantation matérielle optimisée une fois que celle-ci a été déterminée par l'heuristique d'optimisation. Le concepteur se voit ainsi déchargé du codage bas niveau qui constitue une tâche souvent fastidieuse et coûteuse en temps. Ainsi, le temps consacré traditionnellement à l'écriture et au débogage du code RTL de l'application, peut être alors consacré à l'optimisation de l'implantation (en interagissant avec l'heuristique d'optimisation afin de trouver de meilleurs résultats si possible, ce qui peut amener à adapter la taille des grains de l'algorithme, et à minimiser les ressources de l'architecture circuit). Puisque notre méthodologie est basée sur une approche de validation par construction fondée principalement sur la vérification par équivalence des modèles (graphes), le code généré est par conséquent correct par construction. Dès lors, aucun processus supplémentaire de post-validation n'est exigé, la phase de test

des codes peut être fortement diminuée et l'application prototype ainsi obtenue peut être directement utilisée comme produit de série pour l'industrialisation.

7.2 Modèle de génération de code RTL

Nous présentons une nouvelle approche de génération de code RTL qui résout les problèmes de transposition des concepts de spécification comportementale sur les concepts matériels. Les principaux objectifs de notre modèle sont :

- fournir une génération automatique de code à partir d'une spécification,
- que le code produit soit compatible avec les outils de synthèse et de simulation,
- de pouvoir supporter facilement de nouveaux langages de description matérielle sans remettre en cause le processus de génération.

C'est pourquoi nous avons choisi d'utiliser un macro-code intermédiaire indépendant des langages cibles. Ce macro-code utilise des bibliothèques de macros, qui sont dépendantes des langages de description en matériel ciblés. Grâce à un macroprocesseur ce macro-code sera traduit en un langage propre de description matériel par le biais d'appel de bibliothèques de macros. Notre modèle de génération de code RTL consiste ainsi en un processus à deux étapes, dans lequel nous procédons à la génération d'un macro-code intermédiaire avant de transformer ce dernier en code cible à partir d'une bibliothèque définie à priori. Le portage de l'algorithme implanté sur différentes architectures circuits avec des outils de synthèse différents consiste alors à porter ces bibliothèques sur les nouveaux types d'outils et d'architectures, sans remettre en cause le processus de génération ni la spécification de l'algorithme.

Les sections suivantes décrivent les deux phases de notre approche qui permettent de générer le code conformément aux critères que nous venons de décrire :

- La première phase (cf. "Génération de macro" fig.7.1) traduit le graphe d'implantation en un macro-code intermédiaire, qui n'est autre qu'une séquence d'appels de macros. Ce macro-code est générique, c'est-à-dire indépendant du langage cible de description matériel préféré pour chaque type d'outils de synthèse ;
- La deuxième et dernière phase (cf. "Macroprocesseur M4" fig.7.1) traduit le macro-code intermédiaire en code RTL synthétisable par le synthétiseur spécifique à l'architecture circuit cible. Cette traduction est basée sur un macro-processeur complété par un fichier "bibliothèque" de définitions de macros.

7.3 Génération de macro-code générique

Nous rappelons que la construction d'un macro-code intermédiaire générique, plutôt que la génération directe d'un code RTL dans un langage donné et fixe, permet de s'affranchir du choix de ce langage, permettant ainsi une plus grande portabilité du code, ce qui est primordial étant donné l'éventail des possibilités offertes par les avancées technologiques. De plus, cela permet au concepteur de travailler au niveau de spécification qui convient le mieux à ses objectifs.

Avant d'exposer et justifier la structure et la syntaxe du macro-code intermédiaire ainsi que ses règles de construction à partir du graphe d'implantation, nous allons commencer par présenter brièvement les principes d'un macro-processeur.

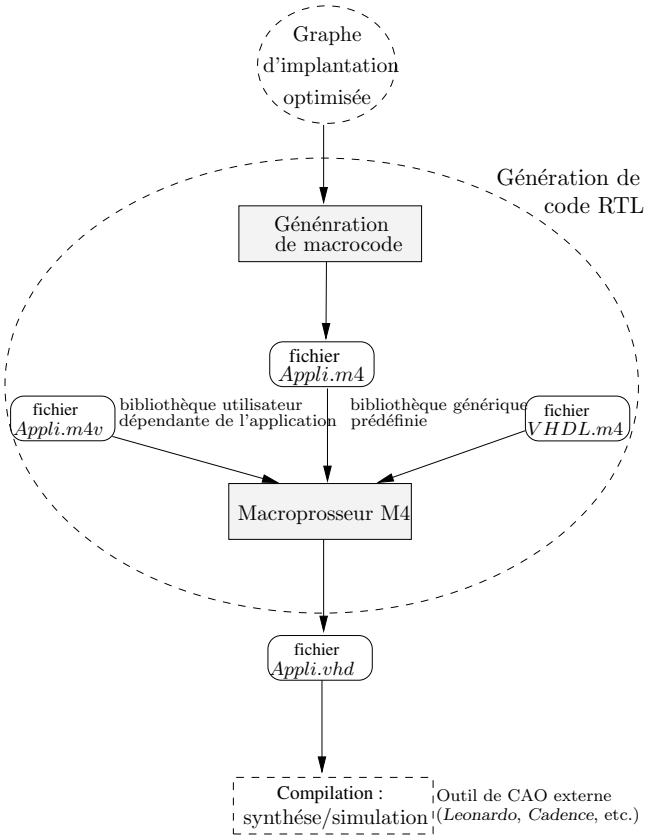


FIG. 7.1 – Modèle de Generation de code RTL (VHDL)

7.3.1 Macro-processeur

Définition

Un macro-processeur est un programme qui prend en entrée un ou plusieurs dictionnaires et qui consomme en entrée une chaîne de caractères (”texte source”), la traite séquentiellement en substituant chaque sous-chaîne (”macro”) dont il trouve une définition dans le dictionnaire par une chaîne correspondante de substitution (”définition de macro”) tirée du dictionnaire qu'il traite à nouveau jusqu'à ce qu'il n'y ait plus de substitution possible, et qui produit en sortie la chaîne de caractères traitée.

Dans un macro-processeur il y a donc un dictionnaire de ”macros” qui associe chaque nom de macro à reconnaître avec une définition. Un appel de macro peut être suivi d'une liste de sous-chaînes arguments qui est substituée aux paramètres formels trouvés dans la définition de la macro. Un certain nombre de macros sont prédéfinies au démarrage du macro-processeur on trouve notamment les macros :

– Include : pour l'appel et le chargement de la bibliothèque de définition de macros,

– Define : pour la création de nouvelles macros.

Nous avons choisi le macro-processeur GNU m4, libre, il a l'intérêt d'être simple mais suffisamment puissant, et d'exister sur toutes les plates-formes, on le trouve en standard sur les systèmes d'exploitation Unix.

Afin de comprendre le code m4 généré nous rappelons ici les principales règles de substitution :

- toute chaîne encadrée par un caractère backquote (‘) à gauche et un caractère quote (’) à droite (et pouvant contenir des backquotes et des quotes balancés) est substituée directement, sans nouvelle tentative de substitution, par la même chaîne sans le premier caractère backquote ni le dernier caractère quote.
  Par exemple, *‘Gen‘’VHDL’* est substituée par *Gen‘’VHDL*,
- les noms de macros ne peuvent être constitués que par des caractères alphabétiques, numériques et du caractère _ (underscore) et ne doivent pas commencer par un caractère numérique (expression régulière [A-Za-z][0-9A-Za-z]*).
  Par exemple, *Gen Xseq1 _1op* sont trois noms de macros possibles,
- les sous-chaînes arguments d’un appel de macro sont séparées par des virgules et leur liste est encadrée entre parenthèses (la parenthèse ouvrante doit être le premier caractère qui suit le nom de la macro).
  Par exemple, *GenVHDL(op,2)* appelle la macro GenVHDL avec deux arguments op et 2,
- la macro define(name, subs) est substituée par une chaîne vide, mais a pour effet de bord de définir une nouvelle macro de nom *name* et de définition *subs*. Pendant une substitution, $n sera substitué par le *n*-ième argument de la macro en cours de substitution (et $0 par le nom de cette macro).
  Par exemple si l’on défini la macro *add* comme suit :*define(‘add’,‘$0q $1+$2’)*, et que l’on donne en entrée du macro processseur la chaîne *add(un,2)*, celui ci va la substituer par *addq un+2*,
- la macro *dnl* est substituée, ainsi que les caractères qui la suivent jusqu’au premier caractère de fin de ligne suivante (inclus), par une chaîne vide (utile pour commenter et formater les sources).

Dénomination des macros

Pour éviter des conflits de noms entre macros générées, les noms fournis par le concepteur de l’application (pour identifier les sommets du graphe de l’algorithme, leurs ports d’entrée et de sortie) sont constitués d’une chaîne de caractères alphanumériques avec initiale alphabétique (comme dans la plupart des autres langages, expression régulière [A-Za-z][0-9A-Za-z]*), mais sans le caractère ’_’ ”underscore” que nous réservons pour constituer les noms des macros à générer automatiquement à partir du graphe d’implantation. Ceci permet de les différencier et d’éviter par conséquent tout conflit avec les premiers.
Par exemple *_Gen_lib()*, *_Gen_entity_const(int,cst,o,1,8,int,0)* sont des exemples de noms de macros générées.

Structure du macro-code généré

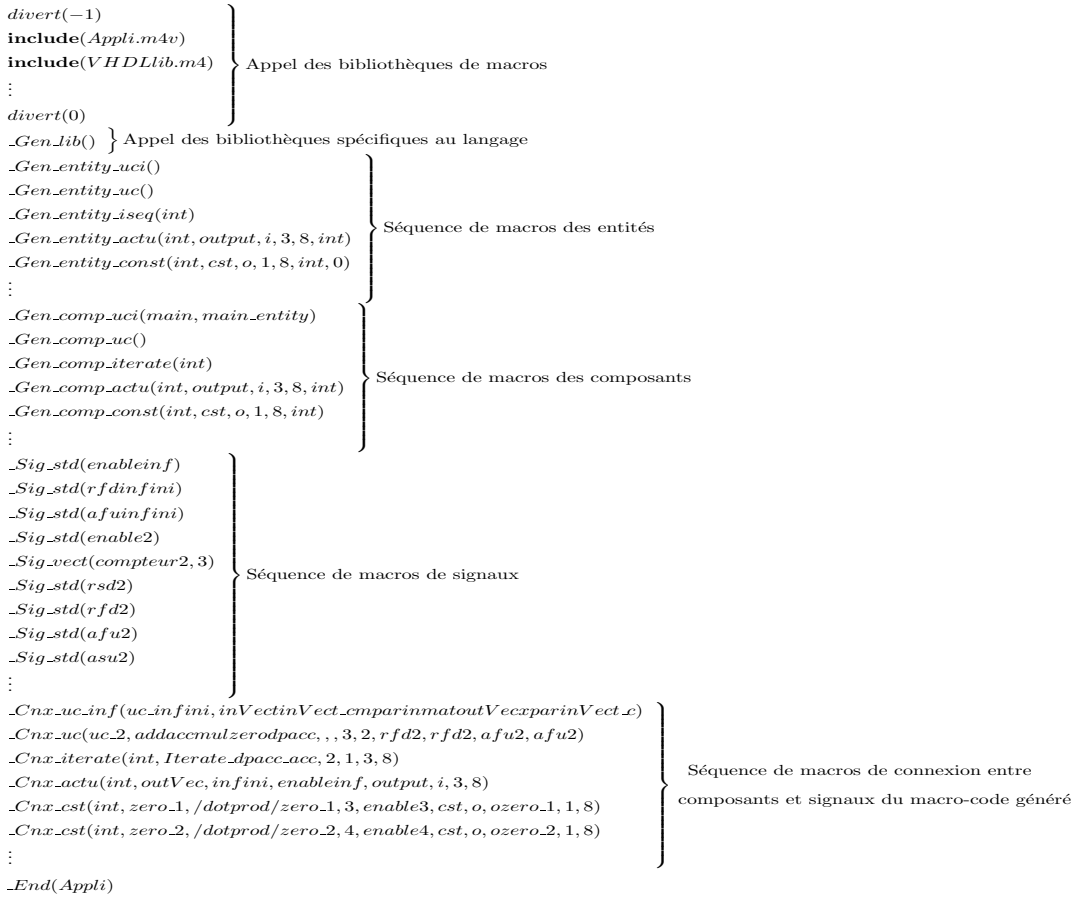
Dans la plupart des langages de description de circuit, les spécifications sont basées sur la même structure : on distingue d’une part des composants à interconnecter, et d’autre part la liste des connexions à effectuer. Nous définissons donc 2 types de macro :

- Les macro de déclaration et de définitions de composants parmi lesquelles nous distinguons
  - Les macros de déclaration de composants du chemin de contrôle
  - Les macros de déclaration de composants du chemin de données
- Les macros de d’interconnexion de ces composants : nous distinguons encore

- Les macros de déclaration de signaux ou "fil"
- Les macros d'interconnexion qui recoivent en argument les composants et les fil/signaux à connecter

La génération de macro-code peut donc être réalisée à partir du graphe d'implantation matérielle G_{im} modélisant les chemins de données et de contrôle de l'architecture circuit correspondante, en remplaçant les sommets du chemin de données du graphe par les macros de définitions des composants correspondants et les arcs de ce graphe, à leur tour, sont remplacés par les signaux d'interconnexion des composants. Les unités et les signaux de contrôle sont également remplacés par les macros de définitions des composants et signaux respectifs. De ce fait, chaque-macro code générée pour chaque algorithme implanté possède la même structure, seul diffèrent les séquences des macros d'opérateurs et de signaux de connexions, qui seront en rapport avec la structure circuit issue de l'implantation et les fonctions de l'algorithme implanté. Chaque fichier de macro-code généré commence donc par les macros include(.. ;) et se termine par _End(...) et il comprend dans l'ordre :

- Des macros d'inclusion des bibliothèques contenant les définitions des macros dans le langage cible de description matérielle du circuit : par défaut une bibliothèque **VHDLlib.m4** de définition des macros décrivant les composants standards en VHDL comme langage de description cible et éventuellement un appel de bibliothèque particulière, par exemple **Appli.m4v** pour un algorithme nommé "Appli", contenant les macros de définitions des opérations spécifiques à l'algorithme et qui ne sont pas incluses dans la bibliothèque prédéfinie par défaut.
- Des macros d'inclusion des bibliothèques spécifiques au synthétiseur cible.
- Une liste de macros de définitions de l'ensemble des composants du circuit matériel issue de l'implantation.
- Une liste de macros de définitions de l'ensemble des signaux du circuit matériel issue de l'implantation.
- Une liste de macros de définitions de l'ensemble des interconnexions entre composants et signaux.



7.4 Transformation du macro-code générique en code RTL cible

C'est la deuxième et la dernière étape du processus de génération automatique de code RTL, elle consiste à transformer le macro-code générique intermédiaire (macros M4) en code cible décrit dans un langage de description en matériel choisi par l'utilisateur.

Pour transformer le macro-code d'une implantation matérielle en code RTL synthétisable dans le langage de description en matériel du synthétiseur du circuit matériel correspondant, nous utilisons le macro-processeur m4.

La façon "standard" d'utiliser ce macro-processeur consiste à lui fournir, sur la ligne de commande, l'ensemble des bibliothèques contenant les définitions des macros, puis à lui fournir le fichier contenant les macros à substituer. Comme chaque fichier de macro-code généré selon notre modèle, commence par des macros d'inclusions include(..;) permettant l'inclusion de l'ensemble des bibliothèques nécessaires à l'interprétation de l'ensemble des macros générées (i.e inclus entre les includes et la macro de fin _End(...)), le code cible sera ainsi obtenu en appliquant la commande m4 directement sur le fichier de macro-code généré.

Le code RTL cible, VHDL par exemple, d'une application donnée 'Appli' peut donc est obtenu à partir du fichier de macro-code généré pour cette application par la commande m4 suivante : $m4 Appli.m4 > Appli.vhdl$.

Après avoir généré le code RTL cible dans le langage de description en matériel désiré (VHDL, Verilog,...)

à l'aide des bibliothèques correspondantes, on utilise la chaîne classique de compilation sur puce via les outils de synthèse, permettant la génération des fichiers d'exécutions nécessaires à la configuration des FPGAs par exemple, ou à la fabrication des ASICs.

7.4.1 Langage du code RTL cible : choix VHDL

De nos jours de nombreux langages de description du matériel (HDL *Hardware Description Language*) tels que Verilog, VHDL, HardwareC, Superlog, etc. existent. Cependant, les deux langages de description de haut niveau Verilog (similaire à C) et VHDL (proche de ADA) s'imposent de plus en plus comme les langages de description matériel les plus communément utilisés.

Le langage de description en matériel VHDL (Very High Speed Integrated Circuit, Hardware Description Language) constitue le fruit du besoin de normalisation des langages de description de matériel (Première norme IEEE 1076-87 en décembre 1987 et standard ANSI en 1988). Auparavant, chaque fournisseur de CAO proposait son propre langage de modélisation (GHDL chez GENRAD, BLM ou M chez Mentor Graphics, Verilog chez Cadence etc...) mais aussi un autre langage pour la synthèse et encore un autre pour le test. Au début des années 80, le ministère de la défense des États Unis (D.O.D) confiait le soin à Intermetrics, IBM et Texas Instruments de mettre au point ce langage. L'objectif était bien sûr d'assurer une certaine indépendance vis à vis des fournisseurs de matériels et de logiciels et ainsi une assurance de maintenabilité des équipements. Le VHDL est donc né du besoin de décrire la fonctionnalité souhaitée indépendamment de la technologie.

Ce langage est conçu avec une sémantique de simulation, s'imposant comme le point d'entrée incontournable des outils de synthèse logique et architecturale. Cependant, la norme qui définit la syntaxe et les possibilités offertes par le langage de description VHDL est très ouverte. Il est donc possible de créer une description VHDL de systèmes numériques non réalisable, i.e non synthétisable par les outils de synthèse, tout au moins, dans l'état actuel des choses. Son usage dans le domaine de la synthèse exige, donc, que le concepteur utilise un sous-ensemble de VHDL structurel dit *synthétisable*, qui correspond à des constructions matérielles identifiables i.e constructions qui décrivent des comportements correspondant à une réalité physique dont les outils actuels de synthèses logiques sont capables de réaliser [**?**] (par exemple décrire une bascule réagissant aussi bien sur front montant que descendant n'est pas réaliste... autre exemple un code VHDL synthétisable ne doit pas comporter de spécification de comportement temporel de type délai qui dépendront généralement des cellules synthétisées et de la cible). Nous énumérons ci-dessous quelques restrictions de ce sous-ensemble dit *VHDL synthétisable* : tous les types de données codables sont admis ; les opérateurs logiques basés sur les types logiques standardisés, les opérateurs arithmétiques utilisant les types *INTEGER* (ou ses dérivés) ou les types vecteurs de bit hérités de paquetages normalisés, les opérateurs de comparaison pour les nombres à représentation binaire et les opérateurs de décalage arithmétiques et logiques sont également acceptés ; tout le jeu d'instruction de VHDL est applicable à la synthèse, sauf l'assertion (*assert*) ; une seule horloge est permise par processus et toutes les constructions VHDL sont acceptables.

Ce langage s'impose comme le langage le plus approprié à nos besoins de génération automatique d'implantation matérielle au niveau RTL à partir d'une spécification algorithmique au niveau comportemental sous la forme d'un GFCDD. En effet, nous devons générer une description structurelle de l'application au niveau RTL, qui soit lisible, modifiable et modulaire, de façon à permettre son implantation intégrale ou partielle par différents concepteurs. Bien évidemment, le code produit doit être

compatible avec les outils de CAO existants (p.ex., *Leonardo*, *Cadence*, etc.). De plus, la description doit être simulable afin de permettre sa vérification fonctionnelle avant l'implantation finale.

Comme nous cherchons surtout à permettre une génération automatique de code, à partir d'une spécification de haut niveau, plus particulièrement à partir de notre modèle de GFCDD et donc du graphe d'implantation matérielle (graphe d'opérateurs interconnectés) correspondant. Nous étions donc amené à développer une bibliothèque de composants VHDL "VHDLLib" (voir Annexe B) qui servira de bibliothèque de substitutions au macroprocesseur GNU m4 en vue de transformer les macros décrivant l'implantation matérielle en code VHDL synthétisable. Cette bibliothèque a été écrite d'après la description des opérateurs de base, des opérateurs de factorisation et des unités de contrôle présentées dans le chapitre 5. La génération de code VHDL peut être réalisée trivialement, en remplaçant les macros-code générées pour les sommets et les dépendances du graphe matériel par les respectifs composants et signaux VHDL.

7.5 Conclusion

Dans le cadre de ce travail, nous avons développé un modèle de génération de code à deux étapes, dans lequel nous procédons à la génération d'un macro code m4 avant de transformer ce dernier en code VHDL à partir d'une librairie VHDL que nous avons définie (VHDLlib.m4). Ce choix de génération de code intermédiaire en vue de la génération du code RTL final en VHDL a été motivé par souci d'indépendance du générateur vis-à-vis du langage de description cible qui peut être aussi n'importe quel autre langage de description en matériel : Verilog ou HardwareC par exemple. Pour ce faire, il suffit seulement de définir et d'intégrer les bibliothèques correspondantes à ces langages cibles afin de permettre de traduire le macro-code intermédiaire généré en code décrit par ces langages sans aucune itération ou remise en cause du processus de génération lui même.

Finalement, comme la génération du code RTL est nécessairement correcte par construction, aucun processus supplémentaire de post-validation n'est exigé et le code RTL généré peut ensuite être directement synthétisé dans un ASIC ou implanté sur un FPGA.

Chapitre 8

Application aux circuits reconfigurables FPGA

Ce chapitre présente brièvement la technologie préconisée comme architecture cible par notre flot d'extension d'AAA (i.e les circuits reconfigurables). L'application de notre flot de conception à ce type de composant sera étudié et les résultats obtenus seront discutés.

8.1 Introduction

Avec l'évolution de la technologie micro-électronique, le concepteur bénéficie de nos jours d'une grande diversité de choix en matière de silicium (ASICs, FPGA, System-On-a-Chip,..) pour la réalisation matérielle de son système.

Ces évolutions technologiques ont particulièrement favorisé la famille des circuits reconfigurables FPGA (Field Programmable Gate Array), qui sont devenus en 10 ans une alternative intéressante aux circuits dédiés ASIC (Application Specific Integrated Circuit).

Aujourd'hui, des circuits intégrés reprogrammables (FPGA) de très grande capacité sont disponibles, et concurrencent sérieusement les circuits intégrés classiques (ASIC).

Les concepteurs s'orientent de plus en plus vers le prototypage de leurs conceptions ASIC au moyen de la technologie FPGA. Ceci est intéressant pour l'évaluation de plusieurs options d'architectures de réalisation pour une application donnée. Le FPGA sert donc, très souvent, à la mise au point, et un fondeur de silicium saura ensuite transcrire ses fonctionnalités en un circuit ASIC figé et a priori de moindre coût lors de production massive.

Nous présentons dans la première partie l'intérêt de ce type de cible par rapport aux composants ASIC traditionnels, puis nous détaillons l'application de notre flot de conception à ce type de composants.

8.2 La cible technologique FPGA x ASIC

Actuellement, dans l'extension AAA/SynDEx, nous utilisons les FPGAs pour l'implantation des parties matérielles. Avant de justifier notre choix des FPGA et non pas des ASICs pour cette extension de la méthodologie AAA, nous tenterons d'abord de rappeler brièvement les caractéristiques de chaque technologie :

- Les ASICs (Application-Specific Integrated Circuit) circuits intégrés spécifiques à une application fournissent précisément la fonctionnalité nécessaire pour une tâche spécifique. Ils sont souvent utilisés pour les parties d'une application bien maîtrisées. Ces circuits sont donc conçus pour une fonctionnalité bien précise et cela leur permet d'être plus petits, moins chers, plus rapide et de consommer moins de puissance. Les ASICs offrent donc les garanties d'un coût réduit en surface (réduction de la taille du système), d'une basse consommation et de meilleurs performances. Par contre la flexibilité, le temps de développement et les possibilités de ré-utilisation sont leurs points faibles. Bien que ces circuits ont fait l'objet de beaucoup d'intérêt pendant la dernière décennie en particulier grâce au développement d'outils de synthèse de haut niveau, dont l'objectif est d'augmenter la productivité de conception de ces circuits, la réduction de leurs temps de conception et développement reste limitée. De ce fait, l'inconvénient majeur des ASICs, si une large diffusion est visée, réside dans les temps de mise à disposition (plusieurs mois) parfois incompatibles avec les exigences du marché. Le développement d'ASICs est donc une solution pour la production en grande série de circuits dédiés, si le temps de développement induit est acceptable, et si son coût, en terme de mise au point est compensé par la taille du marché visé. L'utilisation de circuits ASIC est par contre incontournable si le facteur d'intégration est déterminant pour des raisons de place et/ou de puissance délivrée.
- Les FPGAs (Field Programmable Gate Arrays) ou réseau de portes logiques programmables

permettent de spécialiser une architecture circuit à moindre coût, par programme, sans passer par le processus de fonderie long et coûteux des circuits ASICs. Les avantages de ces circuits programmables sont le temps rapide d'implantation et la possibilité de re-programmation (limitée ou presque infinie dans certains cas). En revanche, leurs inconvénients lorsqu'ils sont comparés aux circuits intégrés, sont le temps d'exécution moins rapide et la densité qui est plus faible. De plus, leur coût d'achat unitaire est plus important que celui d'un circuit ASIC. En raison de leur architecture prédéfinie, les circuits FPGAs sont, pour une application donnée moins performants que les circuits ASICs complètement spécialisés. La popularité des FPGAs peut s'expliquer par leur facilité d'utilisation et leurs performances toujours croissantes. De ce fait, ils sont très adaptées au prototypage quand une émulation rapide d'une spécification est nécessaire. Du fait de leur coût assez élevé par rapport aux circuits ASICs, il est souvent choisi ensuite de produire un circuit complètement dédié, a priori de moindre coût, reprenant les fonctionnalités du circuit programmable, une fois l'application complètement définie et stabilisée. Ainsi, un circuit FPGA peut servir comme circuit de prototypage (ils sont aussi utilisé quand la spécification des fonctions à réaliser risque de changer).

Pourquoi donc choisir les FPGA et non pas les ASIC ? Les ASIC et les FPGA sont également appropriés à l'implantation d'algorithmes de traitement de signal et d'images qui présentent une grande régularité et qui ont besoin de hautes performances de calcul. Pourtant, il est admis généralement que les FPGA présentent tous les avantages de la fonctionnalité sur mesure offertes par les ASIC, tout en évitant les coûts de développements très élevés de ces derniers et leurs incapacités d'être modifiés après leur production. Ceci rend évident notre choix.

Les principaux avantages des architectures à base de FPGA [110] sont donc les suivants :

- le cycle de conception est beaucoup plus court, les FPGAs permettent de transférer une nouvelle application sur silicium en quelques heures alors que la réalisation des circuits intégrés ASICs nécessite des semaines voire des mois ;
- le risque pendant la conception de prototypes en FPGA est faible à cause de son coût négligeable par rapport au coût élevé de la fabrication d'un ASIC.
- la propriété de reprogrammabilité des FPGA face au caractère non-reconfigurable des ASIC ;
- la régularité de la structure des FPGA rend plus facile le développement des outils de synthèse automatique.

Par contre, les inconvénients des architectures à base de FPGA [110], sont les suivants :

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

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

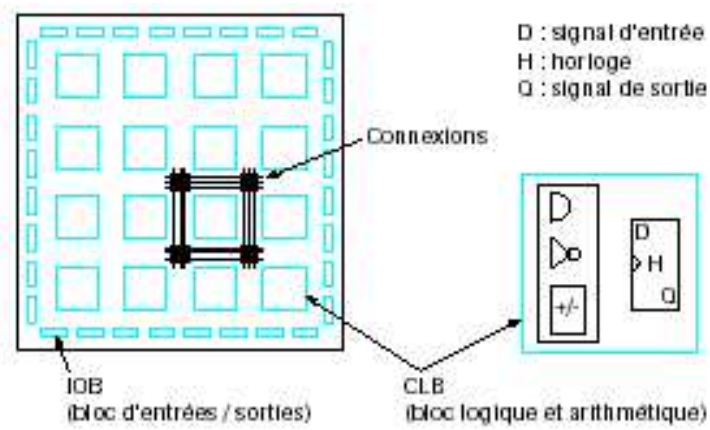


FIG. 8.1 – Schéma d'un circuit reconfigurable (FPGA)

8.3 Qu'est qu'un FPGA

Un FPGA est un réseau (matrice régulière) de blocs logiques combinatoires et séquentiels (cellules CLB, *Configurable Logic Blocks*) placés dans une infrastructure d'interconnexions, voir figure 8.1. Cette infrastructure est entourée par des IOBs (*Input/Ouput Blocks*) servant à communiquer entre l'environnement externe du FPGA et les ressources d'interconnections des CLBs. Un FPGA peut être configuré à trois niveaux :
(1) la fonction des blocs logiques (CLBs),
(2) les interconnexions des blocs logiques,
et (3) les entrées et sorties (IOBs Input/Ouput Blocks).

Cette configuration est réalisée par l'intermédiaire d'une chaîne de bits chargée à partir d'une source externe. En fonction de la façon dont ils sont configurés, les FPGA peuvent être classés comme *configurables* (ils ne peuvent être configurés qu'une seule fois) ou *reconfigurables* (ils peuvent être configurés plusieurs fois). Les FPGA reconfigurables peuvent être *statiques* (la configuration est réalisée avant l'opération du circuit) ou *dynamiques* (le circuit peut être partiellement ou totalement configuré pendant son opération). Les blocs logiques peuvent être des circuits simples, tels que les portes logiques OU, ET, et NON (grain fin), ou des circuits plus complexes, tels que les multiplexeurs, les tables de correspondance et les unités logiques et arithmétiques (gros grain).

Les blocs logiques configurables (CLB) sont les éléments déterminants des performances du FPGA. Chaque bloc est composé d'un bloc de logique combinatoire composé de deux générateurs de fonctions (*notées FG*) à quatre entrées et une sortie et d'un bloc de mémorisation synchronisation composé de deux bascules D de type Flip Flop (*notées FF*). Quatre autres entrées permettent d'effectuer les connexions internes entre les différents éléments du CLB. La figure 8.2 ci-dessous, nous montre le schéma d'un CLB.

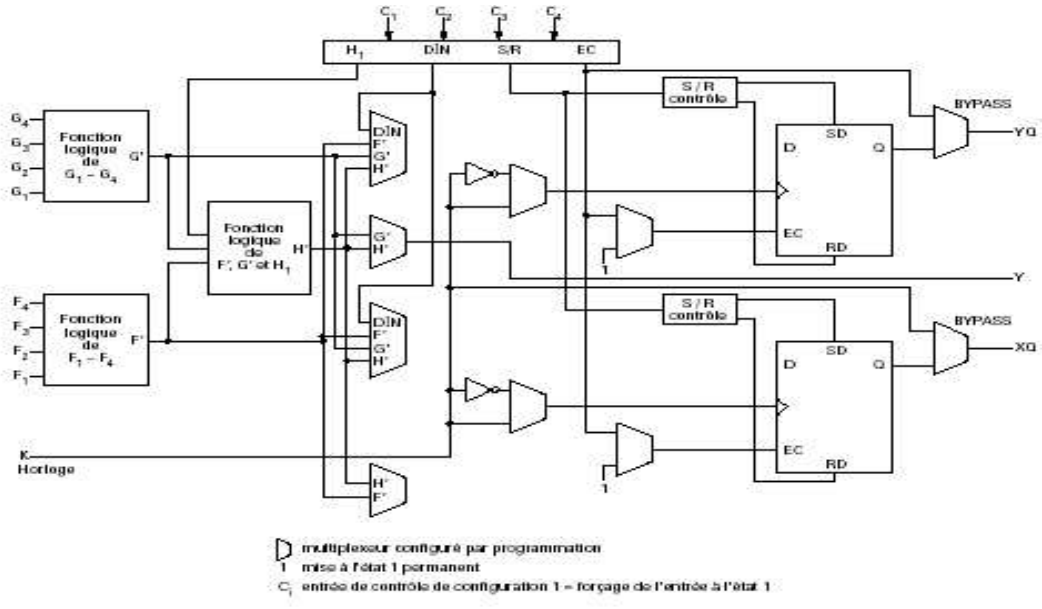


FIG. 8.2 – Schéma d'un bloc logique configurable (CLB)

8.4 Implantation optimisée sur Mono-FPGA

Dans un premier temps, nous nous sommes intéressés aux implantations matérielles sur mono-FPGA. Nous partons ainsi d'un modèle algorithmique de l'application sous forme de GFCDD pour obtenir par transformations de graphes un modèle d'implantation qui respecte les contraintes temps réel de l'application sur un modèle architecturale de type mono-FPGA. Nous rappelons que ce choix de la technologie cible ne remet pas en cause la généralité de l'approche, présentée dans les chapitres précédents, ni les algorithmes appliqués, il permet seulement de préciser les expressions de notre modèle d'estimation de surface une fois que l'architecture cible a été fixée.

8.4.1 Estimation en surface

Comme nous l'avons déjà mentionné dans le chapitre 6, l'estimation de la surface totale ST occupée par le graphe d'implantation d'un algorithme donné, consiste à additionner, à un coefficient près α[1] dû au routage, la totalité de la surface occupée par les différents sommets opérateurs du chemin de données SD ainsi que celle occupée par la logique de contrôle SC (chemin de contrôle).

Dans une implantation sur FPGA, l'estimation de la surface occupée doit être calculée indépendamment en nombre de bascules FF et en nombre de fonctions logiques FG, avant d'être convertie finalement en nombre de CLB occupés grâce à la formule d'approximation : $N_{CLBs} = int(Max(N_{FFs}, N_{FGs})/2)$. Ici comme dans la suite on considère que la fonction $x \mapsto int(x)$ sera l'approximation à l'entier supérieur.

En supposant que l'on connaisse les caractéristiques en surface (nombre de bascules FF et de fonctions FG) de tous les sommets du graphe d'implantation suite à une caractérisation préalable, la surface totale prise par l'implantation d'un algorithme sur un circuit FPGA revient à faire une estimation en

[1] coefficient obtenue de façon empirique en comparant un certain nombre de tests réels, avec les valeurs théoriques.

faisant la somme à un coefficient près de la surface du *data-path* et de la surface du *control-path* en terme de CLBs.

$ST_{CLBs} \leftarrow \alpha \cdot (SD_{CLBs} + SC_{CLBs})$

On obtient alors pour le calcul de la surface du *data-path* SD_{CLBs} l'algorithme de calcul Algo. 12 page 147 et pour le calcul de la surface *control-path* SC_{CLBs} l'algorithme de calcul Algo. 13 page 148.

Surface du *data-path*

Pour calculer la surface du chemin de donnée *data-path*, il suffit d'estimer indépendamment le nombre de bascules FFs occupé par l'ensemble des opérateurs du chemin SD_{FFs}, ainsi que le nombre de fonctions logiques FGs occupé par l'ensemble de ces opérateurs SD_{FGs}, avant de faire une évaluation finale de la surface SD_{CLBs} en nombre de CLBs.

Algorithm 12 Calcul de la surface du *data-path*

Entrée : Le GFCDD étiqueté par les surfaces de ses sommets $G_{al} = (O, D)$
Sortie : La surface SD_{data} occupée par le *data-path*

$SD_{FFs} \leftarrow S_{FFs}(G_{al})$ où $S_{FFs}(G_{al})$ est calculé selon l'algo.4 page 115.
$SD_{FGs} \leftarrow S_{FGs}(G_{al})$ où $S_{FGs}(G_{al})$ est calculé selon l'algo.4 page 115.
$SD_{data} = SD_{CLBs} \leftarrow \dfrac{Max(SD_{FFs}, SD_{FGs})}{2}$

Surface du *control-path*

Comme nous l'avons déjà mentionné en chapitre 6, l'estimation de la surface occupée par le *control-path* se décompose en l'évaluation de la surface occupée par les UCs proprement dites et en la surface des ports logiques nécessaires à la jonction de signaux de requête et d'acquittement de frontières en parallèles et des multiplexeurs nécessaires au contrôle du conditionnement.

Comme à l'intérieur d'une UC le nombre de fonctions FGs est constant, pour estimer le nombre de ressources occupées par la logique interne, il suffit de comptabiliser le nombre de bascules FF nécessaire à l'implantation du compteur interne. Cette évaluation est elle-même très simple ; en effet pour un tel compteur de type "one-hot" il y a autant de bascules que d'états codés, soit le rapport de factorisation f_i de la frontière F_i contrôlée par l'UC en question, qui correspond au nombre de cycles exécutés du coté rapide de la frontière alors qu'un seul a été fait du coté lent.

L'évaluation de la logique externe (les portes ET, et MUX) est elle aussi relativement simple, grâce au graphe de relation de voisinage GVR (de type Max,+,*) dual au graphe de voisinage G_v. Ce graphe représente les relations entre les différentes frontières. C'est un arbre dont les noeuds peuvent être de type "+", les frontières sont en série, de type "*", une frontière est incluse dans une autre ou de type "max", les frontières sont en parallèles. Ce graphe présente de façon explicite la présence de frontières (ou groupes de frontières) en parallèle, qui sont les seules responsables de la présence de signaux à conjonctionner. Lorsque n frontières (ou n groupes de frontières) sont en parallèle (ce qui se traduit sur le G_v par un sommet Max à n sortie) il faut câbler les n signaux de requête (provenant de n UCs de frontières en parallèle) à l'UC de la frontière productrice et les n signaux d'acquittement à l'UC de la frontière consommatrice. Comme pour joindre n signaux il faut une porte ET à n entrées, qui occupe

$\lceil (n-1)/3 \rceil$ fonctions FG, pour évaluer la surface occupée par la logique externe, il suffit de sommer la surface occupée par les portes nécessaires. (Cf. Algo. 13 page 148)

Algorithm 13 Calcul de la Surface du *control-path*

Entrée : Le Graphe de voisinage $G_v = (O', D')$ de l'implantation avec :
- $O' = F_V \cup P_V \cup S_V \cup I_V$
- F_V ensemble des sommets **frontière** ;
- P_V ensemble des sommets **parallèles** (Max) ;
- S_V ensemble des sommets **Série** $(+)$;
- I_V ensemble des sommets **inclusion** $(*)$.

Sortie : La surface totale SC_{CLBs} occupée par le *control-path*

$SCi_{FFs} \leftarrow \sum_{F_i \in F_V} S_{FF}(F_i)$ {surface de la logique interne des UCs en FF}
où $S_{FF}(F_i) = f_i$ si F_i est une frontière de factorisation finie, 0 sinon.
$SCi_{FGs} \leftarrow \sum_{F_i \in F_V} S_{FG}(F_i)$ {surface de la logique interne des UCs en FG}
où $S_{FG}(F_i) = 5$ si F_i est une frontière de factorisation finie, 0 sinon.
$SCe_{FGs} \leftarrow 2 \cdot \sum_{P_j \in P_V} \left\lceil \frac{n_j - 1}{3} \right\rceil$ avec n_j le nombre de sorties du sommet P_j {surface de la logique externe des UCs en FG}
$SC_{CLBs} \leftarrow int(\frac{Max(SCi_{FFs}, SCi_{FGs} + SCe_{FGs})}{2})$

8.4.2 Résultats de l'implantation optimisée

Afin de valider le flot d'implantation proposé, nous avons effectué un ensemble de tests sur des applications diverses. Nous citons en particulier ici le produit matrice vecteur conditionné C-PMV qui a servi d'exemple d'illustration tout au long de ce manuscrit, le filtre de Dériche qui fera l'objet d'une présentation détaillée dans le chapitre suivant.

Le premier tableau Tab. 8.1 présente les résultats des solutions d'implantations optimisées obtenues sous différentes contraintes de temps en utilisant l'heuristique d'optimisation gloutonne décrite précédemment dans le chapitre 6, cf. Algo. 8 page 129. Ces résultats sont présentés en termes, de surface pour les ressources matérielles (nombre de CLBs : Control Logic Blocks) et de latence (en *ns*). Par exemple, pour une contrainte de latence de $1200ns$, l'heuristique opte pour une implantation défactorisée de la frontière FF_2 du C-PMV par un facteur de 2, qui fournit une latence de 864 et nécessite 612 CLBs.

Le deuxième tableau Tab. 8.2 présente les résultats des solutions d'implantations optimisées obtenues sous différentes contraintes de temps en utilisant l'heuristique du recuit simulé décrite précédemment dans le chapitre 6, cf. Algo. 11 page 132. Ces résultats sont présentés en termes, de surface pour les ressources matérielles (nombre de CLBs) et de latence (en *ns*). Par exemple, pour une contrainte de latence de $1200ns$, l'heuristique opte pour une implantation défactorisée de la frontière FF_2 du CPMV par un facteur de 3, qui fournit une latence de 636 et nécessite 593 CLBs.

Les résultats présentés résument les différents graphiques générés par SynDEx-IC suite à son application à l'exemple du C-PMV sur un FPGA *Virtex 300 BG 432* de la famille *Xilinx*. Ces graphiques expriment respectivement la variation du nombres de ressources (CLB) en fonction des contraintes temporelles (figure 8.3) et la variation de la latence du circuit en fonction des contraintes temporelles

Contrainte (ns)	*Implantation*	*Latence* (ns)	*Surface* (CLB)
600	Défact.Totale	433	805
800	Défact.Part. de FF_2 par 3	636	593
1000	Défact.Part. de FF_2 par 2	864	612
1200	Défact.Part. de FF_2 par 2	864	612
1800	Totalement Factorisée	1617	438

TAB. 8.1 – Résultats d'implantation du PMVC sur Mono-FPGA par l'heuristique Gloutonne

Contrainte (ns)	*Implantation*	*Latence* (ns)	*Surface* (CLB)
600	Défact.Totale	433	805
800	Défact.Part. de FF_2 par 3	636	593
1000	Défact.Part. de FF_2 par 3	636	593
1200	Défact.Part. de FF_2 par 3	636	593
1800	Totalement Factorisée	1617	438

TAB. 8.2 – Résultats d'implantation du PMVC sur Mono-FPGA par le recuit simulé

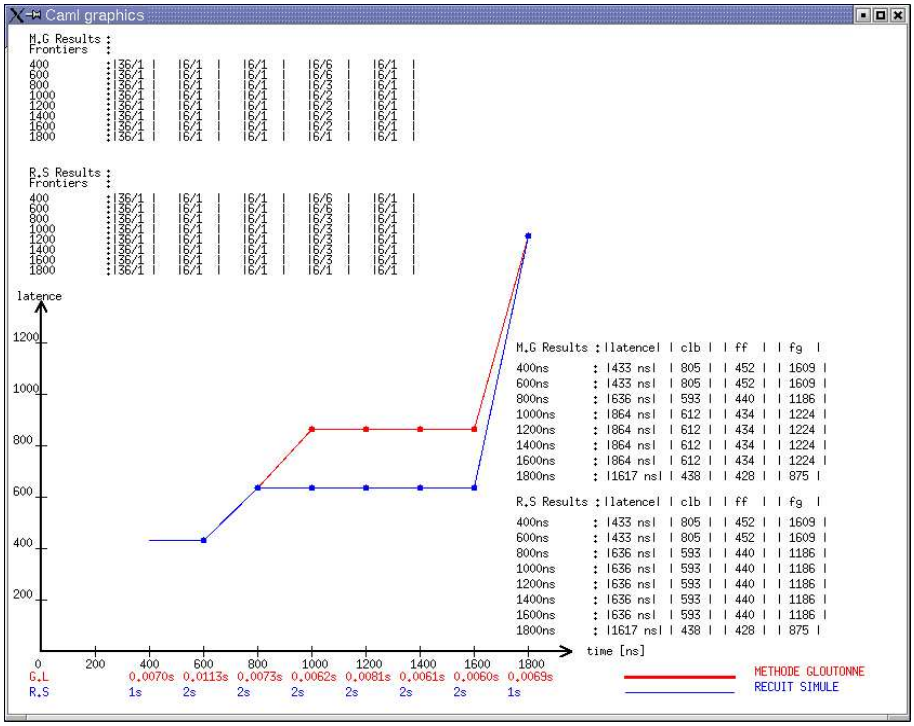


FIG. 8.3 – Variation de la latence en fonction des contraintes

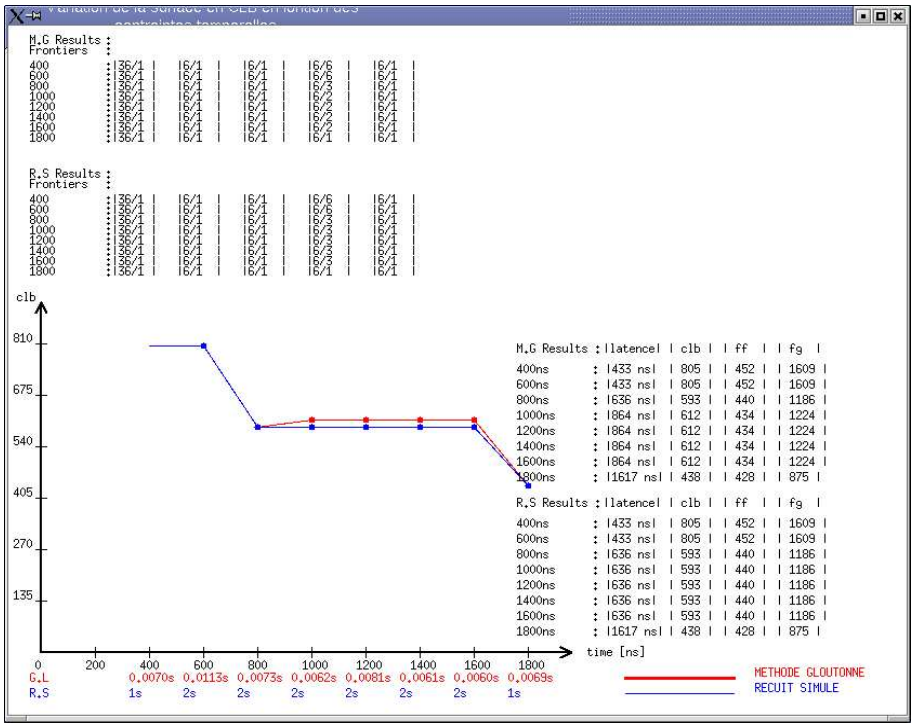


FIG. 8.4 – Variation des ressources en CLBs en fonction des contraintes

(figure 8.4).

8.5 Implantation optimisée sur Multi-FPGAs

Dans le cadre de nos applications cibles, les contraintes temps réel imposées sont parfois tellement importantes que les architectures mono-circuits (mono-FPGA) ne suffisent pas toujours. C'est pourquoi, la seule méthode raisonnable pour atteindre les exigences imposées par ces demandes en puissance de calcul est de recourir à des architectures parallèles c'est-à-dire architectures multi-circuits. En effet, seule une architecture multi-circuits, composée d'un ensemble de ressources de calcul interconnectées entre elles via un réseau d'interconnexions est susceptible de satisfaire les contraintes temps-réels imposées et répondre aux besoins de calcul exigés ce qui revient souvent à utiliser des 'architecture multi-FPGAs'.

Pour pouvoir supporter de telles architectures, nous avons cherché à étendre notre flot d'implantation vers ce type d'architecture en vue d'autoriser l'implantation optimisée d'un algorithme donné sur une architecture multi-FPGAs. Pour ce faire, nous proposons dans cette étude de décrire l'architecture multi-FPGAs sous la forme d'un graphe orienté, l'ensemble des sommets de ce graphe se décompose en deux sous-ensembles, l'ensemble des composants FPGA et l'ensemble des composants Mémoires (SRAM) permettant aux circuits FPGA de stocker leurs données. Chaque sommet - FPGA de ce graphe consomme de données (soit en provenance d'un sommet FPGA soit de sa mémoire SRAM locale) les transforme (traitements) et produit des données (vers d'autres sommets FPGA et/ou écrites dans sa mémoire SRAM locale). Sur la figure 8.5 suivante on présente la structure d'un graphe d'architecture multi-FPGAs avec deux circuits et sans mémoires locales. Il est bon de noter que nous nous sommes

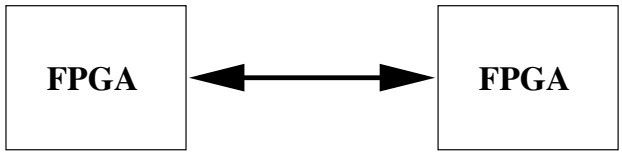


FIG. 8.5 – Graphe d'architecture multi-FPGAs

restreints, dans le cadre de cette étude, à une architecture multi-FPGAs homogènes[2] sans mémoires locales.

8.5.1 Positionnement et formulation du problème de partitionnement

Pour pouvoir implanter un algorithme donné sur une architecture multi-FPGAs, il faut définir et valider des méthodes qui donnent la possibilité de partitionner (distribuer) les différentes opérations du graphe de dépendances de données de cet algorithme sur l'architecture donnée de telle sorte que les contraintes temps réel soient respectées. Cette opération de distribution des différents sommets du graphe décrivant l'algorithme sur les composants reconfigurables de l'architecture est communément appelée *partitionnement*.

La mise en oeuvre de cette distribution (i.e partitionnement) exige le respect d'un ensemble de contraintes imposées, entre autres, par la structure du circuit de type FPGA. Certaines de ces contraintes sont :
- surface maximale, chaque composant FPGA contient un nombre limité de ressources matérielles (CLB, FF et FG) ;
- nombre de PIN d'entrées/sorties ;
- dimensions de la mémoire locale du FPGA, etc.
Toutes ces contraintes ne font qu'accroître la complexité des méthodes de partitionnement.

Ce problème de répartition qu'on se propose de traiter peut être formulé comme suit :
Étant donné une spécification algorithmique sous forme d'un graphe de flot de données $G_{al} = (O, D)$ et la spécification architecturale des composants reconfigurables en termes de taille, de nombre NP de PIN d'I/O, on cherche à trouver un ensemble de partitions $P = P_1, ...P_n$ tel que :
- Chaque partition P_i doit s'ajuster à la surface reconfigurable du composant FPGA correspondant,
- Les communications au niveau de chaque composant doivent respecter sa capacité d'E/S,
- Un certains nombres de contraintes globales de conception par exemple la contrainte sur le nombre de composants de l'architecture, déterminé par l'utilisateur.
- Préservation des dépendances : indique l'ordre d'exécution (toutes les dépendances seront ainsi vérifiées : un ensemble correct de partitions doit assurer le même comportement initial du graphe).

Ainsi, le problème de partitionnement qu'on se propose de traiter est un problème combinatoire même ses versions restreintes sont connues d'être NP-difficile. Il est donc souvent formulé comme un problème d'optimisation sous contraintes, afin d'automatiser l'exploration de l'espace des répartitions possibles. Pour le résoudre, plusieurs techniques ont été proposée dans la littérature.

[2] l'architecture sera composée de la même famille de composants FPGA.

8.5.2 Approches de partitionnement

Pour traiter ce problème, plusieurs approches ont été proposées dans la littérature, certaines procèdent par mouvement des noeuds entre partition *'move-based'* qui sont plus intuitives et très souvent utilisées, d'autres sont basées sur des optimisations combinatoires comme le recuit simulé et les algorithmes génétiques, et celles basées sur des méthodes de résolutions mathématiques comme les méthodes de programmation linéaire en nombres entiers 'Integer Linear Program'. Généralement, ces approches de partitionnement sont regroupées en deux catégories principales : les techniques de bi-partitionnement et les techniques multi-partitionnement.

Les algorithmes de bi-partitionnement ont été utilisés pour les systèmes multi-circuits en les appliquant de manière répétitive sur chaque partition jusqu'à ce que la partition résultante s'ajuste enfin aux contraintes spatiales de l'architecture. Ces techniques de bi-partitionnement procèdent à chaque itération du processus de partitionnement à trouver une répartition optimale de chaque partition en deux partitions distinctes jusqu'à ce qu'elle s'ajuste au contraintes spatiales de l'architecture. Parmi les algorithmes proposés, on cite en particulier l'algorithme KL de Karnighan et Lin [**?**] et sa variante FM proposée par Fiduccia Mattyeyses [**?**] en vue de réduire sa complexité en temps. Ce type de technique a vite montré ses limites, ceci est dû entre autre à leur méthodes de résolution locale du problème. On cite également, l'algorithme K-Way [**?**], qui présente une extension de l'algorithme FM en vue d'aborder le problème en partant initialement d'un partitionnement en n partitions (avec $n \neq 2$) mais qui est basé sur le même principe *'move based'*. Actuellement, on s'oriente de plus en plus vers les méthodes de partitionnement tirées des techniques de multi-partitions qui s'appliquent de plus en plus au problème de partitionnement spatial, plus particulièrement, les techniques d'optimisation combinatoire : le recuit simulé, les algorithmes génétiques, et les méthodes de programmation linéaire.

8.6 Heuristiques de partitionnement proposées

Dans le cadre de notre étude, nous avons proposé deux méthodes de partitionnement du graphe de dépendance de données décrivant l'algorithme de l'application sur les composants reconfigurables de l'architecture multi-FPGAs.

- L'une basée sur une approche d'optimisation par "Recuit simulé", vu les résultats satisfaisants obtenus en Mono-FPGA,
- L'autre basée sur l'approche gloutonne d'optimisation.

Le but de leurs applications est de trouver par défactorisation et partitionnement une partition optimisée de l'ensemble des opérations de l'algorithme sur les composants reconfigurables de l'architecture, laquelle minimise la consommation de ressources matérielles tout en respectant
- la contrainte temporelle, imposée par l'utilisateur ;
- et les contraintes en surface (nombre de CLB) et nombre d'entrées/sorties du circuit FPGA, imposées par les caractéristiques de l'architecture cible.

8.6.1 Approche recuit simulé

À partir du graphe d'algorithme G_{al} sous sa forme factorisée et conditionnée des dépendances de données (GFCDD) et du graphe de voisinage correspondant G_v, l'algorithme de partitionnement basé sur la méthode d'optimisation de "Recuit simulé" fournit le partitionnement correspondant à

l'implantation optimisée du graphe d'algorithme donné en entrée. Avant de présenter et d'étudier les différentes étapes de cet algorithme, on rappelle d'abord la nouvelle modélisation du problème en particulier les nouvelles formes du vecteur d'état décrivant le système et la fonction de coût à optimiser.

Vecteur d'état

L'algorithme à implanter est spécifié sous forme d'un modèle de graphe factorisé et conditionné de dépendances de données entre opérations, noté $GFCDD$. Soit N le nombre de frontières $F_1, F_2, ..., F_N$ de ce graphe. Chaque frontière possède un degré de défactorisation $(d_1, d_2, d_3, ...d_N)$ qui représente le nombre de répétitions du motif répétitif factorisé par la frontière en question (c'est à dire opérations identiques mais opérant sur des données différentes). Soit le vecteur d'état $X = X_1, X_2, ..., X_N$ décrivant l'état de défactorisation/factorisation de chaque frontière du graphe d'algorithme à implanter, chaque variable X_i est à son tour un vecteur dont les éléments correspondent aux différents états de défactorisation possibles de la frontière F_i correspondante. Ainsi à chaque variable X_i du vecteur d'état du graphe d'algorithme correspond un ensemble de variables $(X_{i1}, X_{i2}, ..., X_{id_i})$ qui expriment les états de défactorisation possibles de la frontière correspondante F_i. À un état de défactorisation donné d'une frontière, une et une seule variable sera positionnée à 1 (c'est la variable qui correspond au degré de défactorisation actuel de la frontière). On associe à cette variable qui correspond à un degré de défactorisation donné, l'ensemble des frontières obtenues une fois que la défactorisation, totale ou partielle correspondante a été appliquée. Chacune des frontières résultantes sera placée aléatoirement sur un circuit de l'architecture cible.

Par exemple supposons que la frontière F_2 (dont l'état de défactorisation est représenté par la variable X_2) possède trois états de factorisation (facteur de factorisation c'est-à-dire nombre de répétitions est 3). Alors la variable X_2 possède 3 éléments (X_{21}, X_{22}, X_{23}) correspondant respectivement aux 3 états de défactorisation. Si on choisit de défactoriser F_2 d'un facteur de 2 alors l'élément X_{22} vaut 1 et les autres valent 0. Ainsi chaque variable X_{2i} sera positionnée selon la formule :

$$X_{2i} = \begin{cases} 1 & si \quad degré\ de\ défactorisation = 1 \\ 0 & si \quad degré\ de\ défactorisation \neq 1 \end{cases}$$

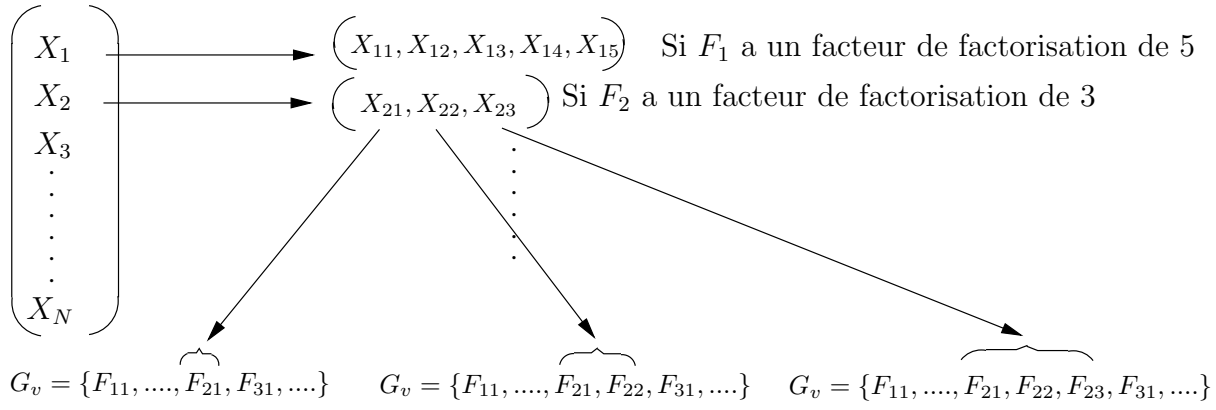


FIG. 8.6 – Vecteur d'état pour l'algorithme de "Recuit simulé" appliqué aux multi-FPGAs

Fonction de coût

Afin d'estimer si une solution obtenue optimise l'implantation donnée, on utilise une fonction de coût $F(X)$ pour laquelle il faut introduire les contraintes en surface et en nombre d'entrées/sorties. Comme le but du partitionnement est de minimiser l'utilisation des ressources (nombre de CLB pour chaque circuit, nombre d'entrées/sorties) en respectant les contraintes temporelles et matérielles, on peut admettre que cette fonction prend comme valeur la somme de la surface occupée par tous les circuits et le nombre total de leurs entrées/sorties si toutes les contraintes sont respectées. Sinon, il faut ajouter des coefficients de pénalité pour chaque valeur qui dépasse la contrainte donnée :

$$F(X) = \begin{cases} S_{tot} + I/O_{tot} & si\ t_{tot} < T_{contr}\ et\ \forall\ S_i < S_{contr}\ et\ \forall I/O\ <\ I/O_{contr} \\ S_{tot} + I/O_{tot} + k(t_{tot} - T_{contr}) & si\ t_{tot} \geq T_{contr}\ et\ \forall\ S_i < S_{contr}\ et\ \forall I/O\ <\ I/O_{contr} \\ S_{tot} + g.(I/O_i - I/O_{contr}) + k(t_{tot} - T_{contr}) & si\ t_{tot} \geq T_{contr}\ et\ \forall\ S_i < S_{contr}\ et\ \exists I/O_i\ \geq\ I/O_{contr} \\ l(S_i - S_{contr} + I/O_{tot} + k(t_{tot} - T_{contr}) & si\ t_{tot} \geq T_{contr}\ et\ \exists S_i \geq S_{contr}\ et\ \forall I/O_i\ <\ I/O_{contr} \\ l(S_i - S_{contr} + g(I/O_i - I/O_{contr}) + k(t_{tot} - T_{contr}) & si\ t_{tot} \geq T_{contr}\ et\ \exists\ S_i \geq S_{contr}\ et\ \exists I/O_i\ \geq\ I/O_{contr} \\ l(S_i - S_{contr} + g(I/O_i - I/O_{contr}) & si\ t_{tot} < T_{contr}\ et\ \exists\ S_i \geq S_{contr}\ et\ \exists I/O_i\ \geq\ I/O_{contr} \\ l(S_i - S_{contr} + I/O_{tot}) & si\ t_{tot} < T_{contr}\ et\ \exists\ S_i \geq S_{contr}\ et\ \forall I/O_i\ <\ I/O_{contr} \\ S_{tot} + g(I/O_{tot} - I/O_{contr}) & si\ t_{tot} < T_{contr}\ et\ \forall\ S_i < S_{contr}\ et\ \exists I/O_i\ \geq\ I/O_{contr} \end{cases}$$

avec :

t_{tot} : le temps d'exécution de l'algorithme après défactorisation,

S_{tot} : la surface totale occupée par tous les circuits après défactorisation,

S_i : la surface occupée par le circuit,

k, l, g : coefficients de pénalité utilisés en cas de non-respect respectivement : des contraintes en temps, surface et nombre d'entrée-sorties,

T_{contr} : la contrainte en temps,

S_{contr} *et* I/O_{contr} : contraintes respectivement en surface et nombre d'entrées/sorties pour toute l'implantation,

I/O_{tot} : le nombre des entrées/sorties total de tous les circuits utilisés pour l'implantation donnée,

I/O_i : le nombre des entrées/sortie du circuit i.

On note que si le nombre des ressources ou d'entrées/sorties d'un circuit (par exemple le circuit i) ne respecte pas les contraintes matérielles, on augmente les coefficients de pénalité correspondants.

8.6.2 Approche gloutonne

Après un partitionnement initial aléatoire, l'algorithme glouton de partitionnement procède par calcul des valeurs optimales des défacteurs des frontières appartenant au chemin critique. Pour chaque frontière candidate à une défactorisation, il calcule sa nouvelle surface, et tente de placer les nouvelles frontières sur le même circuit que la frontière défactorisée si c'est possible, sinon dans les circuits voisins les plus proches triés par ordre croissant de surface libre afin de minimiser autant que possible les communications et la perte en surface. Par la suite il calcule le gain apporté par ce placement et cette défactorisation comme suit :

$G_{new} = (S_{new} - S_{corres}/S_{corres}) + (L_{new} - Ctr)/Ctr$

où S_{new} est nouvelle surface du graphe défactorisé et partitionné

L_{new} est la nouvelle latence du graphe défactorisé et partitionné

Ctr est la contrainte temporelle spécifiée par l'utilisateur

S_{corres} nouvelle surface du graphe défactorisé et partitionné sans perte dû au partitionnement et placement.

Une fois que toutes les frontières ont été traitées de manière équitable, on choisit la solution correspondant au gain minimal. Ce processus est réitéré à chaque fois sur le nouveau graphe résultant jusqu'à ce que la contrainte soit vérifiée.

Algorithm 14 Partitionnement glouton sur Multi-FPGAs

Entrée : Le graphe d'algorithme GFCDD $G_{al} = (O, D)$, son graphe de voisinage $G_v = (O', D')$, la contrainte temporelle Ctr et le nombre de circuits de l'architecture N.
Sortie : graphe d'implantation G_{im} placé sur l'architecture.

Partitionnement initial aléatoire ;
Estimation de la latence courante L_{cur} ;
Identification du chemin critique CH ;
if $L_{cur} > Ctr$ **then**
  Calculer la liste des défacteurs optimaux de chacune des frontières du chemin critique CH
  for all $F_i \in CH$ **do**
    Calculer sa nouvelle surface suite à sa défactorisation SF_{new}
    if $SF_{new} > SC_{free}$ **then**
      Trouver la liste $LSEG$ des segments candidats {I/O direct avec le segment de la frontière en question}
      Trier cette liste par ordre croissant de surface libre
      Placer les sous-graphes images de F dans les éléments de LSEG
      if $LSEG$ est épuisée **then**
        Placer dans les autres segments encore disponibles
      end if
    end if
    Calculer L_{new}, S_{new}, S_{corres}
    Estimer le gain $G_{new} = (S_{new} - S_{corres})/S_{corres} + (L_{new} - Ctr)/Ctr$
  end for
  Choisir le défacteur ainsi que la partition qui correspond au gain minimal
else
  La défactorisation et le partitionnement les plus adaptés ont été trouvés
  Améliorer leur partitionnement
end if

8.7 Conclusion

Nous venons de présenter dans ce chapitre l'application de notre flot d'extension d'AAA aux circuits reconfigurables (FPGAs). L'intégration de ce flot de prototypage sur composants reconfigurables dans l'environnement SynDEx-IC nous a permis de valider notre modèle de conception et de prototyper plusieurs exemples d'applications représentatives du domaine de traitements de signal et d'image : filtre de Dériche, filtre moyenneur,...

Chapitre 9

Développement logiciel : SynDEx-IC

Après cette présentation méthodologique du flot de conception de l'extension d'AAA aux circuits reconfigurables, nous consacrons ce chapitre à la présentation de l'outil d'aide au prototypage rapide d'applications temps réel *SynDEx-IC*, spécialement conçu pour supporter cette extension d'AAA.

9.1 Introduction

Pour répondre spécifiquement à toutes les phases du prototypage rapide d'applications distribuées temps réel de traitement du signal et des images et de contrôle/commande, de la spécification initiale de l'algorithme et de l'architecture jusqu'à l'exécution optimisée temps réel de l'algorithme par les processeurs de l'architecture cible, l'équipe OSTRE de l'INRIA a implanté la méthodologie AAA dans un logiciel interactif de CAO niveau système appelé SynDEx pour Synchronized Distributed Executive.

SynDEx [1] est donc un environnement logiciel graphique interactif de développement pour applications temps réel supportant la méthode AAA d'Adéquation Algorithme Architecture. Il est basé principalement sur l'interactivité avec l'utilisateur grâce à une interface graphique qui permet de spécifier le graphe d'algorithme de l'application (sous la forme d'un graphe flot de données synchrone ou un langage synchrone), le graphe de l'architecture (sous la forme d'un graphe d'opérateurs 'processeurs' décrivant l'architecture cible) ainsi que les caractéristiques de ces graphes. SynDEx est ensuite capable de construire et d'afficher automatiquement le diagramme temporel d'implantation optimisée de l'application grâce à l'heuristique qu'il renferme. Si les caractéristiques de ce graphe d'implantation optimisée sont conformes avec les exigences temps réel de l'application, SynDEx est ensuite capable de générer automatiquement l'exécutif permettant l'exécution de l'algorithme sur l'architecture, libérant ainsi l'utilisateur des tâches lourdes de programmation bas niveau.

Les architectures matérielles supportées actuellement par SynDEx sont des architectures de type "multiprocesseurs hétérogènes" (architectures basées sur des composants programmables à jeux d'instruction figé de type microprocesseurs, microcontrôleur, DSP, station de travail, utilisation d'ASIC dédiés). Pour élargir d'avantage le champs des architectures couvertes par SynDEx et la méthodologie AAA sous-jacente, nous étions amenés, à travers ce travail, à étendre l'architecture cible en vue de couvrir les architectures basées sur des circuits dédiés non programmables, parfois reconfigurables, tels que des ASIC ou des FPGA, qui sont de plus en plus utilisés dans les architectures temps réels embarquées. L'implantation de ce travail d'extension a donné lieu d'une nouvelle version de SynDEx dédiée aux architectures circuits qu'on appelle "SynDEx-IC" [2] (SynDEx pour Integrated Circuit).

SynDEx-IC est donc une extension du logiciel SynDEx de l'INRIA qui permet la synthèse automatique de circuits ASIC et de circuits reconfigurables de type FPGA.

9.2 Récapitulatif de la méthodologie d'extension AAA/SynDEx

Nous rappelons ci-dessous chacune des étapes du modèle de conception proposé, tout au long de ce manuscrit, pour la génération de *netlists* de configuration des architectures basées sur des circuits reconfigurables du type FPGA, à partir d'une spécification algorithmique sous la forme d'un GFCDD. Ce flot de conception représenté par la figure 9.1 a été implanté au dessus du logiciel *SynDEx* qui implante la méthode AAA développée à INRIA–Rocquencourt, résultant ainsi en une nouvelle version de *SynDEx* dédiée aux circuits qu'on appelle *SynDEx-IC*. Dans cette figure, les boîtes grises correspondent aux modules réalisés pour étendre le logiciel *SynDEx* à *SynDEx-IC* (caractérisation matérielle, traduction matérielle, estimation de surface et latence, évaluation de performances, l'optimisation par

[1] logiciel libre disponible sur http ://www.syndex.org
[2] logiciel libre disponible sur http ://www.esiee.fr/a2si/syndex-ic

défactorisation et génération de code RTL "VHDL"). Une fois que le code RTL synthétisable correspondant à l'implantation optimisée a été généré, la *synthèse de circuit* peut être effectuée avec l'aide d'un outil de CAO externe, tel que *Leonardo*, *Cadence*, etc. Les boîtes *spécification algorithmique*, *spécification architecturale* et *contraintes temporelles* représentent les entrées du flot de conception, qui doivent être fournies par le concepteur. La boîte *netlists pour FPGA* correspond aux fichiers de configuration des FPGA générés par l'outil de CAO qui réalise la synthèse matérielle.

1. **Spécification algorithmique** : le concepteur décrit l'algorithme de l'application sous la forme d'un graphe factorisé, éventuellement conditionné, de dépendances de données (noté *GCFDD*), comme décrit dans le quatrième chapitre, par l'intermédiaire de l'interface graphique de *SynDEx-IC* ou à partir d'un fichier *.sdx* obtenu après traduction d'une spécification fait avec un des langages synchrones ;
2. **Spécification architecturale** : le concepteur modélise l'architecture-cible en définissant le nombre de FPGA, le type de FPGA et le type de connexion inter-FPGA, par l'intermédiaire de l'interface graphique de *SynDEx-IC*. Comme dans ce travail, nous nous restreindrons aux architectures mono-FPGA (un seul composant reconfigurable), cette phase de spécification architecturale se résume actuellement à la sélection du type de FPGA utilisé, en termes du nombre de cellules reconfigurables qui le constitue (CLB pour la famille Xilinx) et de E/S ;
3. **Caractérisation matérielle** : Une fois l'architecture spécifiée, il est nécessaire de caractériser l'ensemble des opérations du graphe d'algorithme en termes de surface, en nombre de générateurs de fonction F/G, de CLB et de bascules D (FF), et en termes de latence, en fonction, des caractéristiques du FPGA cible, du codage des données et du nombre de répétitions des motifs factorisés (pour plus de détails voir thèse d'Ailton [27]). Ces valeurs peuvent provenir de bibliothèques ou de mesure ;
4. **Traduction matérielle** : un graphe matériel d'opérateurs modélisant le chemin de données de l'implantation peut être produit automatiquement à partir du GFCDD, en remplaçant les sommets de ce dernier par les opérateurs correspondants (caractérisés en termes de surface et latence) et les transferts de données par les interconnexions des opérateurs. Le contrôleur est aussi caractérisé à partir de l'analyse du graphe des relations de voisinage entre les frontières, comme décrit dans le chapitre 5 ;
5. **Estimation de surface et de latence** : un estimateur de la consommation de ressources matérielles et des performances temporelles analyse le graphe matériel d'opérateurs et estime la surface correspondante au chemin de données et au chemin de contrôle (en nombre de F/G, CLB et FF) et la latence de la traduction matérielle de la spécification algorithmique, cf. décrit dans la section 6.2 ;
   - si la valeur de la latence estimée est inférieure à la contrainte temps réel, nous pouvons effectuer l'implantation matérielle détaillée de la spécification algorithmique ;
   - sinon, il faut réaliser l'optimisation par défactorisation de la spécification algorithmique ;
6. **Optimisation par défactorisation** : l'optimisation par défactorisation de la spécification algorithmique consiste à choisir une transformation à appliquer sur le GFCDD, à l'aide de notre heuristique de défactorisation, en fonction des contraintes temporelles fournies par le concepteur, des performances temporelles (latence) et de la consommation de ressources matérielles (surface)

calculées par l'estimateur pour cette spécification. Il s'agit principalement de dérouler plus ou moins les boucles : plus une boucle est déroulée, plus il y a de parallélisme, plus il est possible de diminuer la latence, mais en contrepartie la surface nécessaire à son implantation augmente. La transformation choisie (taux de défactorisation ou taux de déroulage) cherche à réduire la latence de l'implantation matérielle correspondante et à minimiser l'augmentation de la consommation de ressources matérielles provoquée par la défactorisation. Après cette étape, il faut répéter les étapes correspondantes à la *traduction matérielle*, à l'*estimation de surface et latence* et à l'*évaluation des performances* et, si nécessaire, la *défactorisation*, jusqu'à obtenir une implantation qui respecte les contraintes temporelles et minimise la surface, ou jusqu'à la défactorisation totale de la spécification algorithmique, cf. décrit dans la section 6.3. Enfin, étant donné que le prototypage rapide est notre objectif, l'heuristique utilisée par défaut dans *SynDEx-IC* est celle basée sur l'algorithme glouton décrit en chapitre 6, section 6.3.4. Cependant, selon le niveau d'optimisation exigé et le temps disponible, l'utilisateur peut choisir l'exécution d'une des heuristiques (gloutonne, recuit simulé) de SynDEx-Ic. ;

7. **Implantation matérielle** : l'implantation matérielle détaillée produit un graphe matériel élargi (graphe d'implantation), composé du graphe d'opérateurs généré par la *traduction matérielle* de la spécification algorithmique (correspondant au graphe *data-path*), plus les circuits nécessaires à la synchronisation des transferts de registres (unités de contrôle). Ces circuits, ajoutés au graphe matériel d'opérateurs, composent la "partie contrôle" (ou *control path*) de l'implantation matérielle, comme décrit dans le chapitre 5 ;
8. **Génération de code RTL** : une fois le choix de l'implantation optimisée effectué, chaque opérateur du graphe correspondant à l'implantation matérielle serait traduit en un module RTL sous la forme d'un composant d'une bibliothèque VHDL. La génération de code consiste à interconnecter ces composants pour obtenir un code RTL synthétisable (VHDL), qui correspond à l'implantation optimisée de la spécification algorithmique définie par le concepteur, cf. chapitre 7 ;
9. **Synthèse des circuits** : le code VHDL structurel synthétisable généré est utilisé comme entrée d'un outil de CAO externe (p.ex., *Leonardo*) pour effectuer la synthèse de l'architecture-cible. Dans ce travail, l'outil de synthèse génère les *netlists* nécessaires à la configuration des FPGAs.

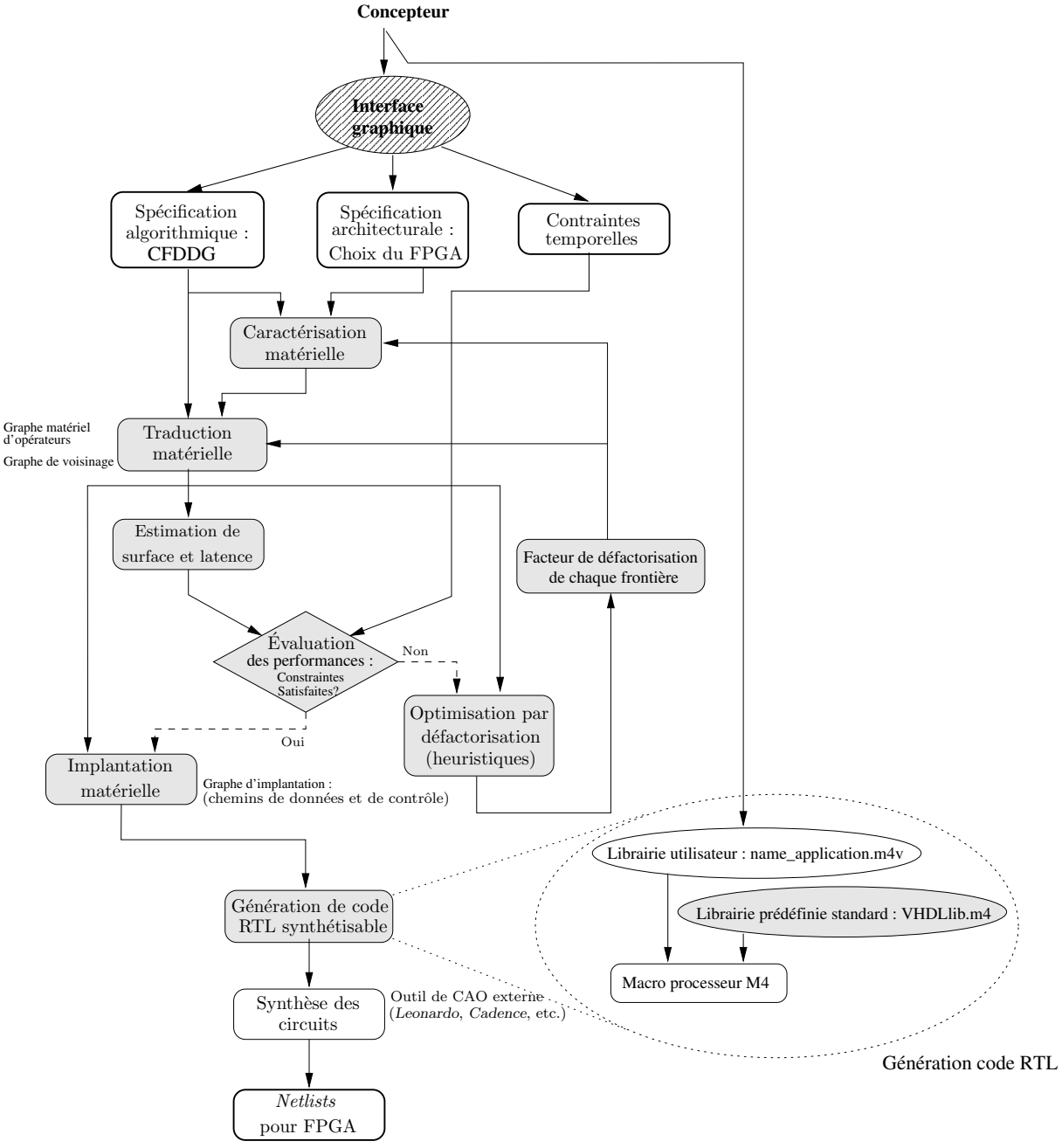


FIG. 9.1 – Modèle de conception d'architectures mono-FPGA dans AAA/SynDEx-IC

9.3 Présentation de l'environnement logiciel SynDEx-IC

L'outil SynDEx sur lequel est implémenté la méthodologie AAA ne supportant pas les composants reconfigurables, nous avons donc étendu la méthodologie AAA et développé une nouvelle version SynDEx-IC en partant de la version SynDEx6.6.1 de l'INRIA dans le but de couvrir ce type de composants. Notre travail a consisté dans un premier lieu à dégager de SynDEx les parties réutilisables dans le cas de l'implantation sur circuits, et à développer des modules spécifiques. Nous avons utilisé le même langage utilisé pour le développement de la version originale de SynDEx, c'est à dire le langage Caml. Ce langage fonctionnel, développé à l'INRIA, est particulièrement adapté à la manipulation d'objets mathématiques comme les graphes qui constituent la structure de base de notre flot d'extension d'AAA aux circuits.

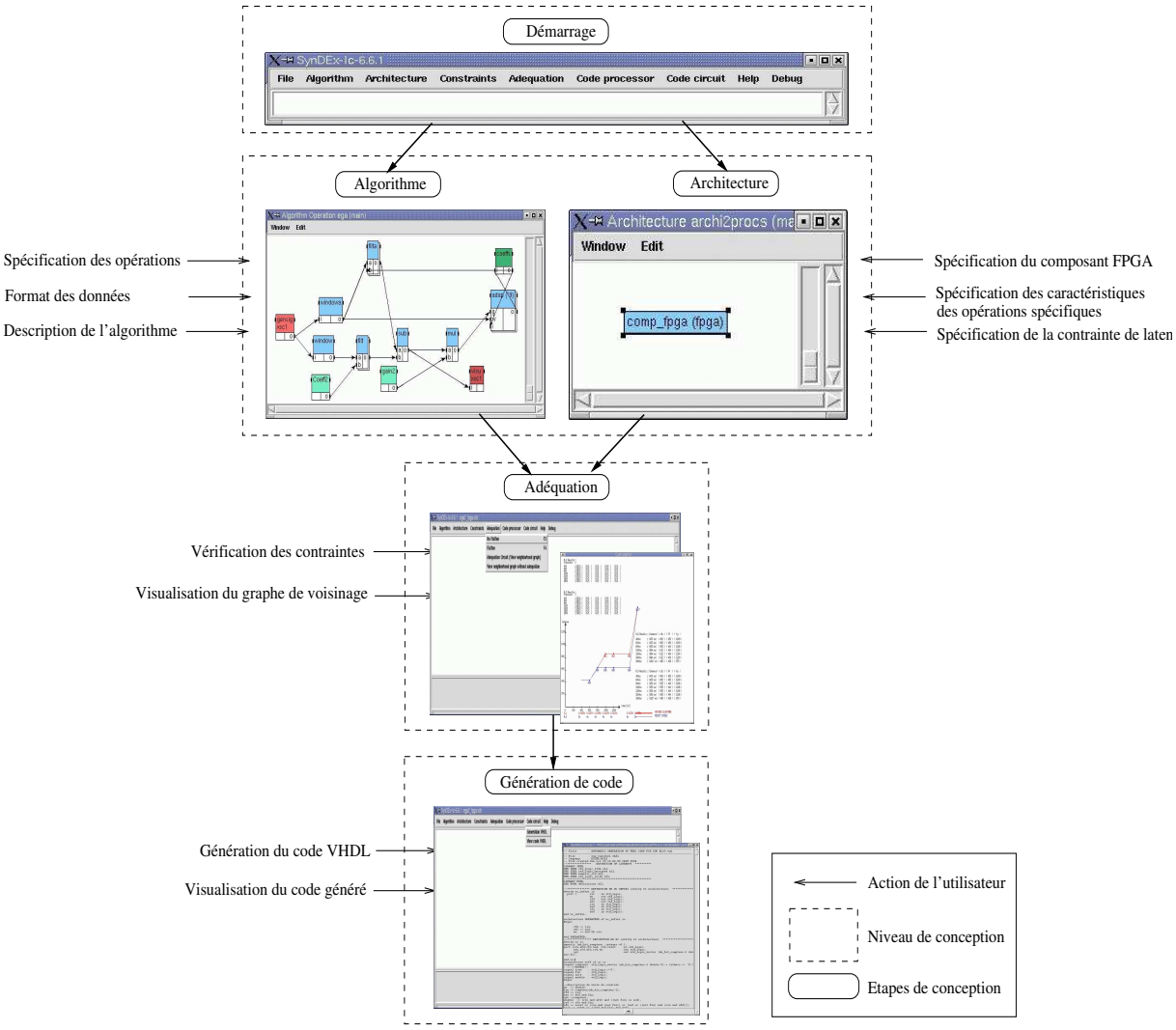


FIG. 9.2 – La conception sous SynDEx-IC

L'IHM, développée en Tcl/Tk fournit une interface graphique utilisateur pour manipuler le coeur

Caml de SynDEx-IC. Elle permet à l'utilisateur de spécifier graphiquement l'algorithme et l'architecture, de lancer et d'interagir avec l'heuristique d'optimisation en permettant de visualiser les résultats (prédiction des performances) de l'heuristique, et d'afficher graphiquement le graphe de voisinage de l'implantation et enfin de lancer la génération automatique de code.

La conception sous SynDEx-IC comprend différentes étapes, allant de la spécification de l'algorithme et de l'architecture (Mono-FPGA) jusqu'à la génération de code VHDL synthétisable. Les différentes grandes étapes de cette conception sont présentées dans la figure 9.2 ci-dessous :

Tout d'abord l'utilisateur doit spécifier, à travers l'interface graphique de SynDEx-IC, le graphe d'algorithme de son application à l'aide d'un graphe flot de données conditionné factorisé hiérarchique, le type du composant FPGA de son architecture ainsi que la contrainte de performances souhaitée.

Après avoir spécifié et caractérisé son graphe d'application, il suffit de presser un bouton pour exécuter l'heuristique d'adéquation basée sur la défactorisation. Ainsi en fonction du niveau d'optimisation exigé et du temps disponible, l'utilisateur dispose de deux types d'heuristiques appartenant à des familles différentes. La durée de cette étape dépasse rarement quelques minutes. Les résultats d'évaluation des performances seront visualisés sous formes de courbes de performances dans une seconde fenêtre. Une fois que la prédiction de SynDEx-IC est satisfaisante, il suffit de presser un bouton pour qu'il génère automatiquement le code RTL en VHDL correspondant aux choix d'implantation de l'heuristique. Le code ainsi généré servira par la suite à la configuration de l'architecture d'implantation. L'utilisateur peut enfin visualiser le code généré avant de procéder à sa simulation sur un outil de synthèse externe (Leonardo Spectrum par exemple).

La spécification algorithmique étant le point initial d'interaction de l'utilisateur avec l'environnement SynDEx-IC, à partir duquel vont s'enchaîner les différentes étapes de conception, nous illustrons ici à travers la spécification de l'exemple du produit matrice-vecteur conditionné traité dans les chapitres précédents comment spécifier aussi bien des algorithmes factorisés que des algorithmes conditionnés sous SynDEx-IC.

- **Pour spécifier les parties factorisées d'un GFCDD :** l'approche de spécification hiérarchique dans SynDEx-IC permet de spécifier chaque sommet du graphe par un sous-graphe permettant ainsi de développer des modules réutilisables et de faciliter la compréhension d'un algorithme par encapsulation des détails fins. Ce processus de spécification hiérarchique fournit en plus un moyen simple et élégant d'exprimer la répétition, chaque niveau de hiérarchie peut à son tour délimiter une frontière de factorisation différente marquant les limites du motif répétitif correspondant, comme il est montré sur la figure 9.3.
- **Pour spécifier les parties conditionnement :** l'idée consiste à construire un sommet auquel ne correspond pas un sous graphe mais un ensemble de sous-graphes étiquetés par des valeurs entières symboliques. Selon la valeur d'un entier reçu sur un port spécial du sommet, c'est un sous-graphe différent qui est exécuté. Sur l'exemple de la figure 9.4 l'entrée x de l'opération conditionnée c définit deux sous-graphes correspondant chacun à une valeur 0 ou 1.

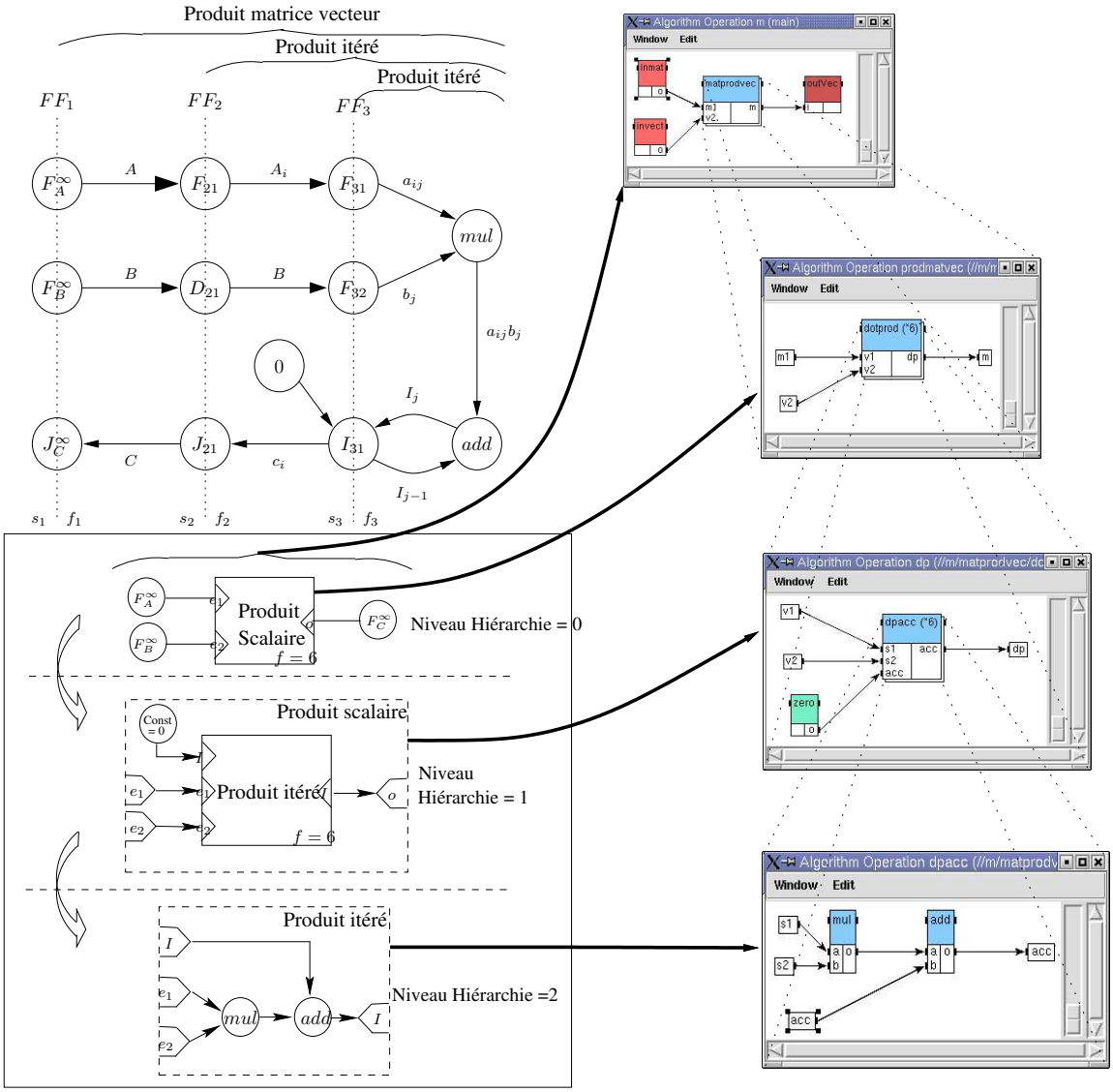


FIG. 9.3 – Hiérarchisation et factorisation

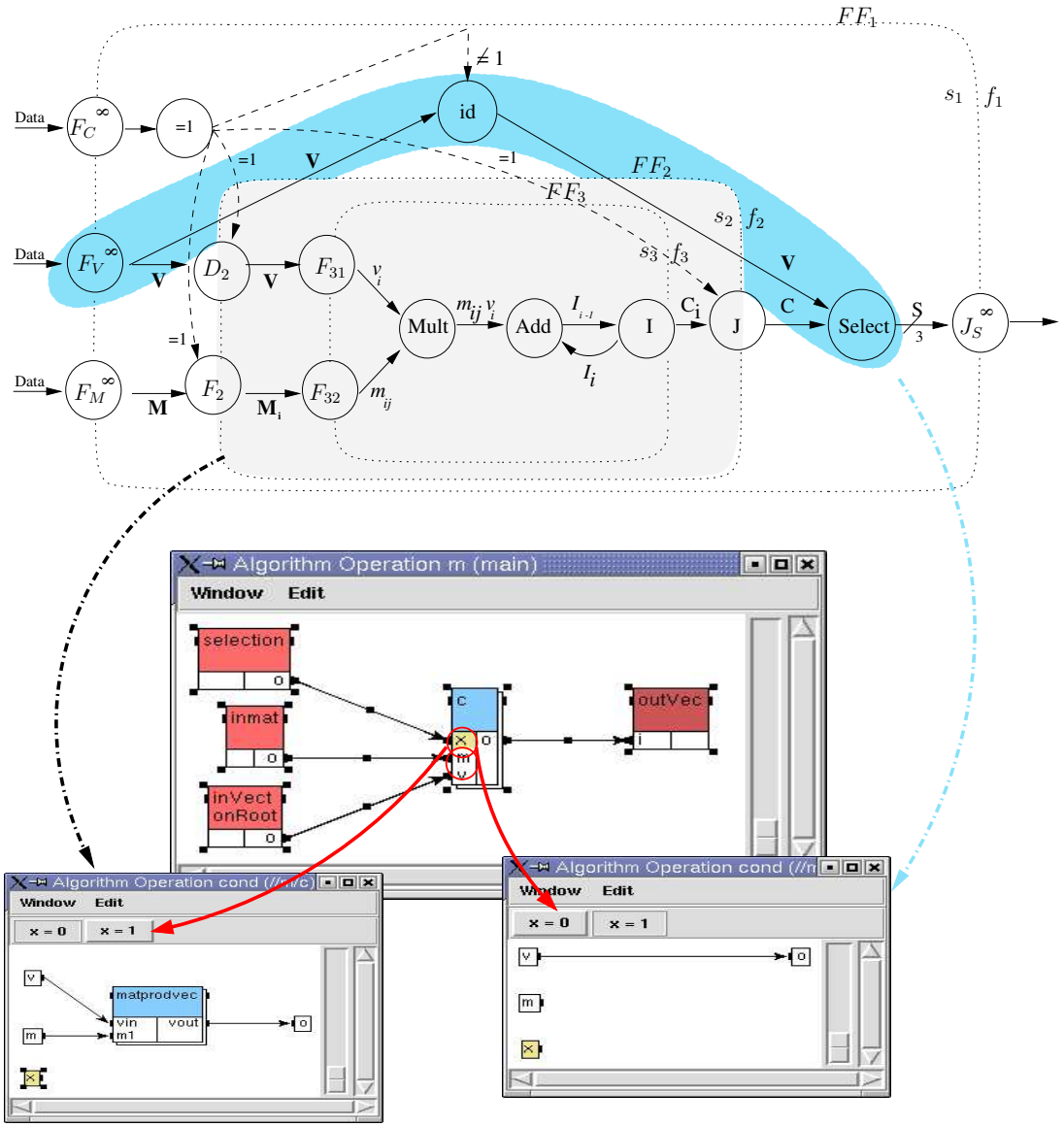


FIG. 9.4 – Hiérarchisation et conditionnement

9.4 Exemple d'application : filtre de dériche

Pour illustrer notre extension AAA/SynDEx, nous avons choisi un exemple concret d'algorithme récursif de détection de contour dû à Rachid Dériche. Ce filtre, en particulier sa version optimisée proposée par Garcia Lorca, est très utilisé en traitement d'images pour sa qualité de détection, pour la possibilité de faire varier simplement sa résolution et pour sa complexité de traitement plus faible que celle des filtres non récursifs. Généralement l'application de ce filtre nécessite de parcourir l'image à plusieurs reprises. De plus, sa complexité calculatoire intrinsèque a conduit plusieurs équipes de recherche à réaliser des implantations FPGA ou ASIC (Application Specific Integrated Circuit) dans le cadre d'une mise en oeuvre temps réel de ce type de filtre. Le choix de cet exemple illustre donc parfaitement nombre de problèmes liés aux implantations temps réel.

9.4.1 Présentation du filtre

Le filtre optimisé de Garcia-Lorca décompose le filtre de Deriche en une cascade de filtres lisseurs causaux et anti-causaux pour chaque direction de l'image, comme décrit sur la figure9.5. Il comporte donc globalement deux grandes étapes :

- un lissage vertical et horizontal
- une dérivation

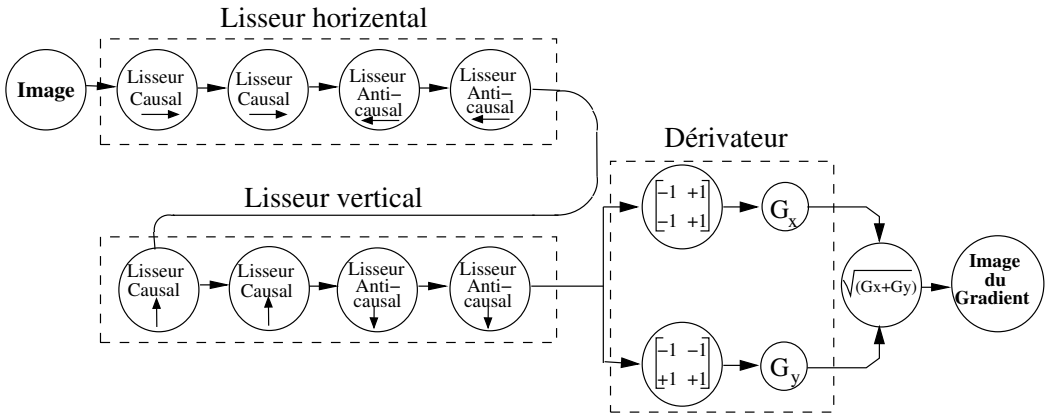


FIG. 9.5 – Filtre cascade récursif du 1^{er} ordre de Deriche

La figure suivante (voir Fig.9.6) donne le schéma global de l'implantation de ces étapes :

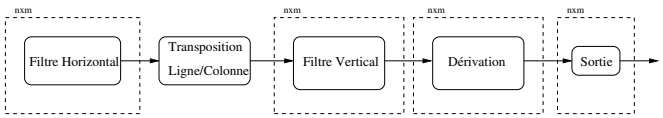


FIG. 9.6 – Architecture du filtre de dériche

On commence par effectuer le lissage horizontal ligne par ligne. Ensuite un câblage permet de transposer les lignes en colonnes. On applique par la suite le lissage vertical et la dérivation.

Le lissage

C'est le même horizontalement et verticalement. Pour un lissage horizontal sur une ligne, on effectue tout d'abord deux passes d'un lisseur du premier ordre causal dont l'équation est :

$$y(i) = (1-\gamma)x(i) + \gamma y(i-1)$$

$x(i)$ étant le $i^{ième}$ pixel de la ligne en cours
$y(i)$, le $i^{ième}$ pixel filtré
Puis deux passes d'un filtre anticausal dont l'équation est :

$$y(i) = (1-\gamma)x(i+1) + \gamma y(i+1)$$

Le filtrage horizontal global (donc deux passes pour le filtre lisseur causal et deux pour le filtre lisseur anticausal) peut être vu comme 4 passes du même filtre causal à la condition d'intervertir les indices des éléments d'une ligne. C'est ce que réalise le bloc inversion sur la figure 9.7 :

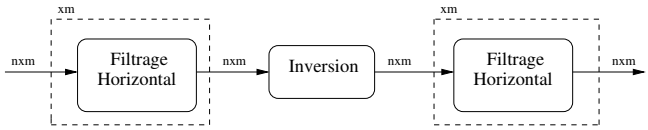


FIG. 9.7 – Filtre Horizontal

On procède de même pour le filtre vertical sur les colonnes de l'image.

Le dérivateur

Il procède par l'application des deux filtres de dérivation R_h et R_v dont les masques sont :

$$R_h \begin{pmatrix} -1 & 1 \\ -1 & 1 \end{pmatrix} \quad et \quad R_v \begin{pmatrix} -1 & -1 \\ +1 & +1 \end{pmatrix}$$

Le résultat est ensuite normé. La norme communément utilisée est la norme euclidienne.

Spécification des différents éléments du filtre

Par la suite, nous présentons la spécification algorithmique sous forme de GFCDD des principaux éléments constituants le filtre : le filtre lisseur causal d'ordre un et le dérivateur.

Filtre lisseur d'ordre Un Tout d'abord, un bloc **Premier** permet d'isoler le premier élément entrant $x(0)$ et de fournir les $n-1$ éléments restants dans un même vecteur. Au cycle i , l'élément $x(i)$ est dans l'*Iterate*"I" et l'élément $x(i+1)$ passe dans le fork "F". On applique alors les opérations du filtre lisseur sur ces deux éléments *i.e* on fait :

$$(1-\gamma)x(i) + \gamma x(i+1)$$

Le résultat de l'opération passe ensuite dans le même temps dans le *Join* "J" et est réutilisé dans l'*Iterate* "I" pour le cycle suivant (voir Fig.9.8).

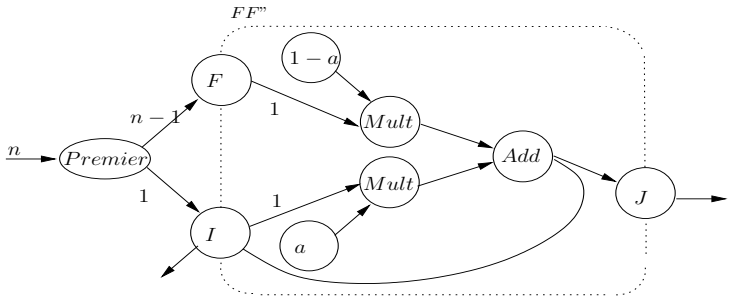


FIG. 9.8 – Filtre lisseur d'ordre un

Dérivation Le schéma général du filtre Fig. 9.9 ressemble beaucoup dans sa conception au filtre lisseur d'ordre un : on se sert des *Iterate*"I" pour réutiliser l'élément que l'on vient de traiter.

Parce que cela est possible, on va appliquer les deux masques de dérivations R_h, R_v sur la matrice de pixel suivante :

$$\begin{pmatrix} x_{ij} & x_{ij+1} \\ x_{i+1j} & x_{i+1j+1} \end{pmatrix}$$

en réalisant en même temps les opérations :

$$\begin{pmatrix} -x_{i,j} & x_{i,j+1} \\ -x_{i+1,j} & x_{i+1,j+1} \end{pmatrix} et \begin{pmatrix} -x_{i,j} & -x_{i,j+1} \\ x_{i+1,j} & x_{i+1,j+1} \end{pmatrix}$$

On calcule ensuite la norme euclidienne des résultats obtenus. La figure suivante présente le schéma général du dérivateur :

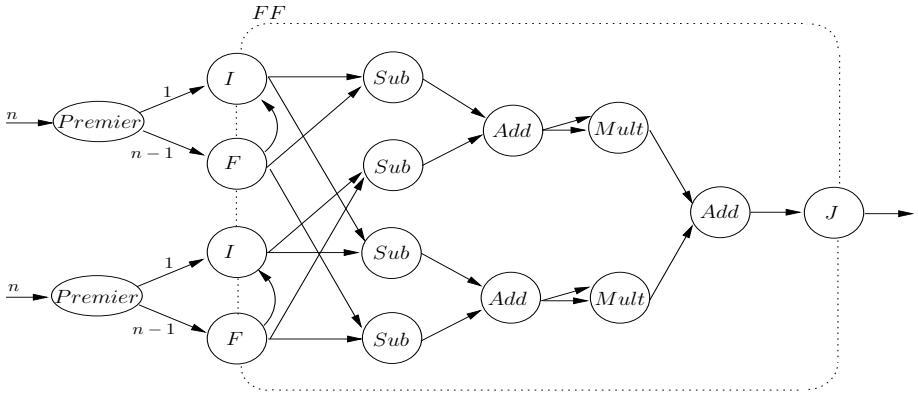


FIG. 9.9 – Le dérivateur

9.4.2 Prise de graphe sous SynDEx-IC

La prise de graphe sous SynDEx déduite du GFCDD décrivant le filtre est donnée par les figures suivantes 9.10 et 9.11. La figure 9.10 présente en haut la spécification globale du filtre suivie de la

spécification hiérarchisée des différents éléments constituants le filtre horizontal. Sur la figure 9.11 : On peut voir en dessous la partie du graphe correspondant au dérivateur.

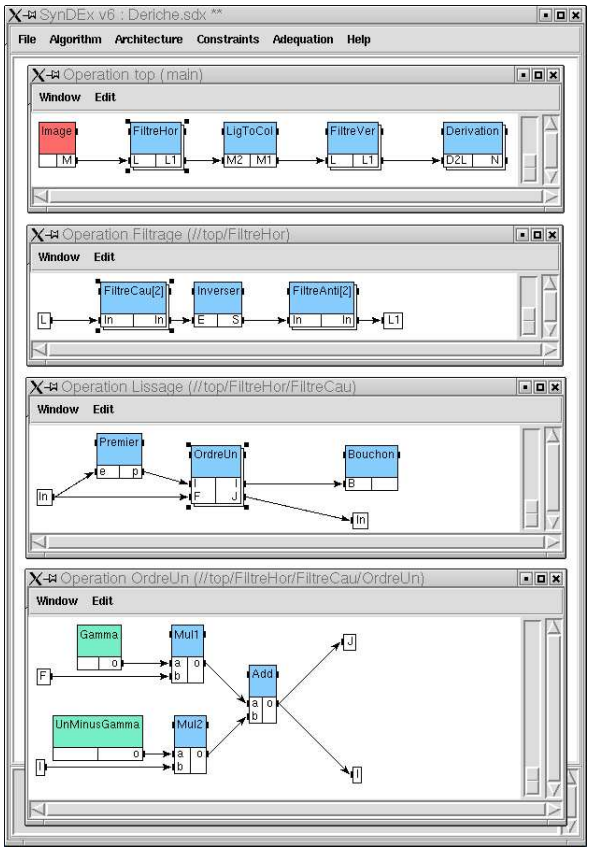


FIG. 9.10 – Spécification SynDEx du Filtre de Dériche : Filtrage

9.4.3 Résultats

Nous avons testé l'implantation du filtre appliqué à des images de 16*16 pixels sur des architectures FPGA *Virtex 300 BG 432* sous différentes contraintes de temps. Le Tab. 9.12 présente les solutions obtenues par l'heuristique d'optimisation (décrite précédemment) sous certaines contraintes. Par exemple, pour une contrainte de latence de $1400ns$, l'heuristique opte pour une implantation défactorisée de la frontière FF du dérivateur par un facteur de 3. Ces résultats présentés en termes de surface pour les ressources matérielles (nombre de CLBs) et de latence (en ns) montrent que les solutions les plus défactorisées permettent de réduire la latence mais augmentent en contrepartie la consommation en ressources. De ce fait, selon les contraintes imposées, les latences des solutions obtenues évoluent ainsi par palier Fig. 9.13.

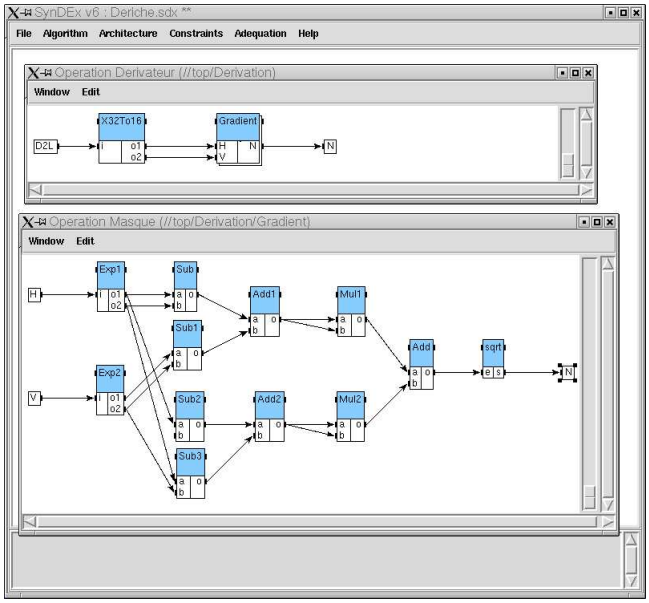


FIG. 9.11 – Spécification SynDEx du Filtre de Dériche : Dérivation

Implementation	*Contrainte* (ns)	*Surface* (CLB)	*Latence* (ns)
Totalement Factorisée	2000	1406	1961
Défact.Part.FF par 2	1800	1346	1453
Défact.Part.FF par 3	1400	1444	1235
Défact.Part.FF par 4	1200	1384	1162
Défact.Tota.FF	1000	2344	944

FIG. 9.12 – Résultats d'implantation sur FPGA

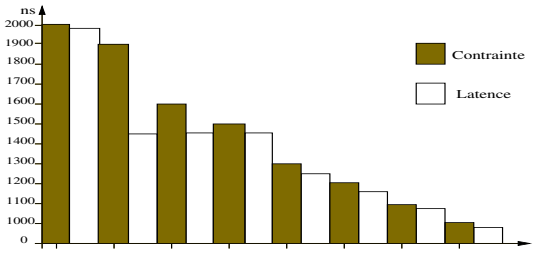


FIG. 9.13 – Optimisation sous contraintes de temps

9.5 Conclusion

Nous venons de présenter dans ce chapitre l'environnement logiciel SynDEx-IC qui concrétise notre flot d'extension d'AAA pour le prototypage rapide et l'implantation optimisée d'applications sur circuits reconfigurables. Nous avons explicité les différentes étapes permettant de générer avec SynDEx-IC le code RTL correspondant à l'implantation optimisée d'une application sur un circuit reconfigurable.

À partir du même graphe (GFCDD) spécifiant l'application, il est possible actuellement de générer une implantation optimisée pour une architecture multicomposant en utilisant SynDEx ou une synthèse optimisée pour circuit intégré reconfigurable FPGA en utilisant SynDEx-IC. Les deux outils étant fondés sur le même modèle d'application et les mêmes principes cela facilite le travail du concepteur.

On note également que SynDEx-IC est juste un outil intermédiaire qui génère du code uniquement pour un FPGA. Des travaux sont en cours pour étendre AAA/SynDEx à des architectures mixtes en intégrant AAA/SynDEx-IC dans AAA/SynDEx afin de supporter dans un même outil la génération d'exécutables distribués temps réel pour les composants programmables (processeurs) et non

pro-grammables (ASIC, FPGA), ainsi que la synthèse de chacun des composants non programmables à partir du code RTL généré. Cette extension est envisageable en vue d'unifier les aspects programmables et non programmables des modèles, de manière à poser formellement (le plus objectivement possible) le problème de la conception conjointe logiciel/matériel.

Chapitre 10

Conclusions et perspectives

La complexité grandissante des applications temps réel embarquées de contrôle commande et de traitement du signal et des images requiert une puissance de calcul de plus en plus forte. L'implantation de telles applications est dans la plupart des cas un exercice délicat à maîtriser, pour la simple raison que la satisfaction de cette demande croissante en puissance de calcul et des contraintes temporelles, nécessite l'adoption des architectures mixtes qui ne contiennent pas seulement des processeurs standards (DSP, RISC etc) mais aussi des processeurs spécialisés sous forme de circuits synthétisés (circuits dédiés (ASIC)ou circuits reconfigurables (types FPGA)). Ces architectures mixtes sont adoptées dans le but d'effectuer de manière performante certaines parties du traitement et améliorer de ce fait la performance globale du système implanté.

Par conséquent, il existe de nos jours un réel besoin en méthodologies de développement haut niveau qui soient associées à des environnements logiciels efficaces afin d'aider les développeurs d'applications à implanter conjointement et à développer rapidement ces applications temps réel distribuées et optimisées (i.e. qui respectent les contraintes temps réels et minimisent la taille des architectures). Pour répondre à ce besoin, plusieurs méthodologies et outils de conception conjointe ont été développés pour assister les développeurs d'applications temps réels dans la phase d'implantation.

Cependant l'état d'art entrepris sur la conception conjointe matériel/logiciel nous a permis de constater que, parmi les différents outils universitaires et commerciaux de conception conjointe existants, aucun n'est capable de résoudre de façon satisfaisante les problèmes de spécification indépendante de l'implantation, de validation et d'implantation conjointe tout en offrant un flot complet et unifié comportant une spécification graphique, le partitionnement entre la partie matérielle et la partie logicielle, la prédiction des performances, la génération de code exécutable distribué et optimisé pour la partie programmable et la génération de code synthétisable et optimisée pour la partie reconfigurable, tout ceci sous des contraintes de temps réel et de ressources. Ceci est notre objectif à long terme en étendant la méthodologie AAA d'Adéquation Algorithme Architecture supportée par *SynDEx* aux circuits reconfigurables. AAA/SynDEx vise tous ces objectifs mais ne couvre pas actuellement les architectures dédiées (composants non programmables). Les travaux effectués dans le cadre de cette thèse présente donc une extension à de tels circuits.

Étant donné que nos applications cibles exigent le recours à des architectures mixtes multicomposants, nous avons cherché à travers ce travail à étendre la méthodologie AAA afin de supporter les composants non programmables en particulier ceux qui sont reconfigurables tel que les FPGAs. Avec de

tels composants, nous avons rencontré deux nouveaux problèmes. Premièrement, nous devons générer le code RTL incluant la synthèse automatique du chemin de données et du chemin de contrôle à partir de la spécification algorithmique sous sa forme factorisée et conditionnée des dépendances de données. Deuxièmement, puisque les ressources sont limitées dans les composants reconfigurables (i.e le nombre de cellules reconfigurables (CLB) pour les FPGAs), il est préférable de faire des implantations factorisées des parties répétitives des algorithmes (nous avons à créer des "boucles" en silicium afin de diminuer le nombre de ressources requises) mais ceci peut conduire à une augmentation de la durée d'exécution (non respect des contraintes temporelles). De plus, comme il est fondamental de veiller à ce que le code implanté reflète exactement la sémantique et les propriétés d'ordre partiel de la spécification initiale, le processus complet de génération de code RTL depuis la spécification algorithmique a été modélisé, validé et formalisé en termes de transformation de graphes, en présentant à chaque fois les règles adéquates de construction et en faisant la preuve de la conservation des propriétés du graphe d'algorithme à chaque étape.

Le flot de spécification-implantation que nous avons proposé est par conséquent un flot continu sans rupture depuis la spécification jusqu'à l'implémentation finale ce qui facilite d'une part la conception sans erreur (pas de déformation ou perte d'information liées à une transcription de modèle) et d'autre part la traçabilité.

Ce flot de génération automatique de circuits synthétisables optimisés à partir de spécifications algorithmiques exprimées sous forme de graphes factorisés et conditionnés de dépendances de données, est capable de synthétiser toute la partie de contrôle des parties répétées du graphe (boucles) et des parties conditionnées. On note également que l'approche de génération du contrôle préconisée est une approche de contrôle réparti plutôt qu'un contrôle centralisé ce qui permet de rapprocher les unités dépendantes et ainsi minimiser les surcoûts dû au routage et contrairement aux approches classiques notre chemin de contrôle n'est pas une machine à états monolithiques mais au contraire un ensemble de circuits combinatoires simples synthétisés automatiquement.

D'autre part, comme il est impossible d'explorer toutes les possibilités d'implantations optimisées dans des temps raisonnables, le flot proposé renferme des méthodes approchées basées sur des heuristiques qui permettent de rechercher automatiquement l'implantation qui réalise le bon compromis entre l'espace (occupé par l'application sur le FPGA) et le temps (temps d'exécution de l'application). Plus précisément, il s'agit d'explorer les boucles et boucles imbriquées de façon à trouver pour chaque boucle le taux de défactorisation (taux de déroulage) qui permet de faire tenir l'application dans le circuit (contrainte de surface) tout en respectant la contrainte de latence spécifiée. Nous avons développé une heuristique gloutonne qui peut être couplée avec l'heuristique de voisinage de recherche locale de type recuit simulé. L'heuristique gloutonne essaie de trouver rapidement l'implantation qui respecte les contraintes données. Cependant comme ce type d'heuristique peut passer à côté de solutions meilleures, l'autre heuristique est une heuristique de voisinage de recherche locale de type recuit simulé qui bien que plus lente explore plus de solutions ce qui permet des optimisations plus fines, en utilisant comme solution initiale l'implantation construite avec l'heuristique gloutonne.

Le résultat de l'heuristique est ensuite transformé en un code VHDL directement synthétisable pour être exécuté sur le composant FPGA cible ou bien pour être simulé.

Nous avons implanté ce flot dans le logiciel "SynDEx-IC" que nous avons développé à l'ESIEE à partir du noyau SynDEx de l'INRIA. SynDEx-IC est donc un logiciel d'aide au prototypage rapide

d'applications temps réel conçu spécialement pour supporter notre flot d'extension d'AAA. Il permet la synthèse automatique de circuits ASIC et de circuits reconfigurables de type FPGA.

Même si, au cours de cette thèse, un flot complet de spécification-implantation a été adopté et présenté, une série des travaux sont encore à considérer en vue de prolonger ce travail. Parmi ces travaux, certains font partie de la liste non exhaustive suivante : actuellement il est nécessaire d'ajouter aux circuits non programmable ainsi synthétisés un circuit supplémentaire effectuant les communications inter-composant avec d'autres composants (programmables ou non programmables). Nous envisageons de synthétiser aussi automatiquement ces communications pour chaque composant non programmable. A plus long terme on envisage aussi d'effectuer du partitionnement logiciel/matériel automatique optimisé en fusionnant les heuristiques servant à la génération d'exécutifs et celles servant à la synthèse de circuit (génération de code RTL).

Bibliographie

[1] Stephen A. Edwards. Languages for Digital Embedded Systems. Kluwer, Boston, Massachusetts, 2000.

[2] Donald E. Thomas and Philip R. Moorby. The Verilog Hardware Description Language. Kluwer, Boston, Massachusetts, fourth edition, 1998.

[3] IEEE Computer Society, 345 East 47th Street, New York, New York. IEEE Standard Hardware Description Language Based on the Verilog HardwareDescription Language (1364-1995), 1996.

[4] Douglas L. Perry. VHDL. McGraw-Hill, New York, third edition, 1998.

[5] Stephen A. Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems : Formal models, validation, and synthesis. Proceedings of the IEEE 85(3) : 366-390, March, 1997.

[6] J. Plantin and E. Stoy, "Aspects on System-Level Design", in 7th International ACM/IEEE Workshop on Hardware/Software Codesign - CODES'99, pp. 209-210, may 1999.

[7] L. A. Cortés, P. Eles, Z. Peng. A Survey on Hardware/Software Codesign Representation Models. SAVE Project Report, Dept. of Computer and Information Science, *Linköping* University, *Linköping*, Sweden, June 1999.

[8] Edward A. Lee, "Embedded Software," Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002 ou Technical Memorandum UCB/ERL M001/26 University of California, Berkeley, CA 94720, July 12, 2001.

[9] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli. A Formal Specification Model for Hardware/Software Codesign In Proceeding of International Workshop on Hardware-Software Codesign, October 1993. ou Technical Report UCB/ERL M93/48, dept.EECS, University of California, Berkeley, June 1993.

[10] Harel, Statecharts : A visual formalism for complex systems. In Science of Computer Programming, vol. 8, pp. 231-274, June 1987.

[11] F.Vahid, S.Narayan, and D.Gajski, SpecCharts : A VHDL front-end for embedded systems, IEEE-Transactions on Computer-Aided Design of Integrated Circuits and Systems, 14(6)694-706, 1995.

[12] C.A. Petri, Kommunikation mit Automaten, Ph.D. Thesis, Institut for Instrumentelle Mathematik, Bonn, Germany, 1962.

[13] Y. Atamna, "Réseaux de Petri temporisés stochastiques classiques et bien formés : définition, analyse et application aux systèmes distribués temps réel," Rapport LAAS No. 94418, Thèse de Doctorat, Université Paul Sabatier, Toulouse, Octobre 1994.

[14] E. Stoy : A Petri Net Based Unified Representation for Hardware/Software Co-Design, Licentiate Thesis, LiU-Tek-Lic 1995 :21, Dept. of Computer and Information Science, Linkp̃ing University, 1995.

[15] J. Peterson, Petri Net Theory and the Modeling of Systems, Englewood Cliffs, NJ : Prentice-Hall, 1981.

[16] Z. Peng and K. Kuchcinski. Automated Transformation of Algorithms into Register-Transfer Level Implementations. IEEE Transactions on Computer-Aided Design of Integrated Circuitsand Systems, 13(2) :150-166, Feb. 1994.

[17] G. Dittrich, Modeling of Complex Systems Using Hierarchical Petri Nets, Codesign : Computer-Aided Software/Hardware Engineering, J. Rozenblit and K. Buchenrieder, Eds. Piscataway, NJ : IEEE Press, 1995, pp. 128-144.

[18] Lee, Dongho ; Kim, Chongsang Extended Timed Petri Net Models for Performance Analysis of Communication Protocols. In : J. Korea Inf. Sci. Soc. (South Korea), Vol. 15, No. 2, pages 141-148. April 1988.

[19] C.A. Hoare. Communicating Sequential Processes. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[20] B.P. Zeigler, H. Praehofer, et T.G. Kim. Theory of Modelling and Simulation : 2nd Ed. : Integrating Discrete Event and Continuous Complex Dynamic Systems, Academic Press 2000.

[21] W.-T. Chang, S. Ha, and E. A. Lee. Heterogeneous simulation mixing discrete-event models with data ow. Journal of VLSI Signal Processing, 15 :127 144, 1997.

[22] E. A. Lee. Modeling Concurrent Real-time Processes Using Discrete Events, Invited paper to Annals of Software Engineering, Special Volume on Real-Time Software Engineering, 1998. Also UCB/ERL M98/7 Technical Report, Dept.EECS, University of California, Berkeley, March 1998.

[23] C. G. Cassandras. Discrete Event Systems : Modeling and Performance Analysis. Boston,MA : Aksen Associates, IRWIN Publications, 1993.

[24] E. A. Lee, Modeling Concurrent Real-Time Processes using Discrete Events, Technical Report UCB/ERL M98/7, Dept. EECS, University of California, Berkeley, March 1998.

[25] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language in Science of Computer Programming, 2002 ou Technical Memorandum UCB/ERL M001/33 University of California, Berkeley, CA 94720, September 15, 2001.

[26] GAJSKI, Daniel D., VAHID, Frank. Specification and design of embedded hardware-software systems. *IEEE Design & Test of Computers*, v.12, n.1, Spring 1995, p.53-67.

[27] A. DIAS. Contribution à l'implantation optimisée d'algorithmes bas niveau de traitement du signal et des images sur des architectures mono-FPGA à l'aide d'une méthodologie d'Adéquation Algorithme Architecture. Thèse de Doctorat, Université Paris XI Orsay. Juillet 2000.

[28] DIAS A., LAVARENNE C., AKIL M., SOREL Y., " Optimized implementation of real-rime image processing algorithms on field programmable gatearrays ", Fourth International Conference on Signal Processing, Beijing, Chine, 1998.

[29] A. Benveniste and P. Le Guernic, Hybrid Dynamical Systems Theory and the SIGNAL Language, IEEE Trans. on Automatic Control, vol. 35, no. 5, pp. 525-546, May 1990.

[30] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The Synchronous Data Flow Programming Language LUSTRE, Proc. of the IEEE, vol. 79, No. 9, 1991, pp. 1305-1319.

[31] G. Berry and G. Gonthier, The Esterel synchronous programming language : Design, semantics, implementation, Science of Computer Programming, vol. 19, no. 2, pp. 87-152, 1992.

[32] A. Vicard. Formalisation et optimisation des systèmes informatiques distribuées temps réel embarqués. Thèse de Doctorat, Université de Paris-Nord XIII, Juillet 1999.

[33] T. Grandpierre. Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réels optimisés. Thèse de doctorat, Université de Paris-Sud ORSAY, Novembre 2000.

[34] T. Grandpierre. Un nouveau modèle générique d'architecture hétérogène pour la méthodologie AAA. Dans Actes des Journées Francophone sur l'Adéquation Algorithme Architecture (JFAAA'2002), pages 6-9, Monastir Tunisie, 16-18 Décembre 2002.

[35] T. Grandpierre, C. Lavarenne, Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx - Rapport de Recherche INRIA No.3476, Août 1998.

[36] A. DIAS, M. AKIL, C. LAVARENNE, Y. SOREL. Adéquation Algorithme Architecture appliquée aux circuits reconfigurables. 4ème édition des Journées Adéquation Algorithme Architecture en traitement du signal et images, CEA/LETI, Saclay, France , Janvier 1998.

[37] A. DIAS, M. AKIL, Y. SOREL, C. LAVARENNE. Vers la synthèse automatique de circuits à partir de graphes algorithmiques factorisés, conférence. AAA'2000, INRIA Rocquencourt, Janvier 2000.

[38] Y. Sorel. Massively parallel systems with real time constraints : the "Algorithm Architecture Adequation" Methodology. In Proc. Massively Parallel Computing Systems, the Challenges of General-Purpose and Special-Purpose Computing Conference, Ischia Italy, May 1994.

[39] Y. Sorel. Méthodologie A3 d'adéquation algorithme architecture. Chapitre 5 du livre, Méthodes et architectures pour le TSI en temps réel, par D. Demigny, 2001.

[40] Y. Sorel. Real-time embedded image processing applications using the AAA methodology. In Proc. of the IEEE Int. Conference on Image Processing, November 1996.

[41] C. Lavarenne, Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel Traitement du signal, vol. 14 n° 6, pp. 569-578, 1997.

[42] C. Lavarenne, O.Seghrouchni, Y. Sorel, M.Sorine. The SynDEx Software Environment for Real-Time Distributed Systems, Design and Implementation, European Control Conf. Grenoble France, 1991.

[43] J.B. Dennis. First Version of a Dataflow Procedure Language, Lecture Notes in Computer Sci., Springer-Verlag, vol.19, pp.362-376, 1974.

[44] J. B. Dennis., "Data flow computer architecture", Final Report MIT/LCS/TR-385, October 1987.

[45] R. Ernst. Codesign of embedded systems : Status and trends. IEEE Design and Test of Computers, pages 45-53, April-June 1998.

[46] G. De Micheli, R. K. Gupta. Hardware/software Co-design. IEEE, Vol. 85 No. 3,pages 349-365, March 1997.

[47] R. Gupta, G. De Micheli. Hardware-software Co-synthesis for Digital Systems, IEEE Design and Test of Computers, 1993.

[48] G. De Micheli. Computer-Aided Hardware-Software Codesign. IEEE Micro pages 10-16, 1994.

[49] J. Plantin, E.Stoy. Aspects on system-level design. Proceedings of the Seventh International Workshop on Hardware/Software Codesign, pages 209-210, 1999 (CODES '99).

[50] V. D. Zivkovic, P. Lieverse. An overview of methodologies and tools in the field of system-level design, Lecture Notes In Computer Science Embedded processor design challenges : systems, architectures, modeling, and simulation-SAMOS archive pages 74 - 88, 2002.

[51] G. Bosman. A Survey of Co-Design Ideas and Methodologies. Master's Thesis the Vrije Universiteit in Amsterdam, Août 2003.

[52] W.H. Wolf. Hardware-Software Co-Design of Embedded Systems, Proceedings of the IEEE, vol. 82, no. 7, pp. 967-989, Jul. 1994.

[53] C. Kuttner. Hardware-Software Co-Design Using Processor Synthesis, IEEE D&T of Computers, pp. 43-53, Fall 1996.

[54] A. A. Jerraya, M. Romdhani, P. L. Marrec, F. Hessel, P. Coste, C. Valderrama, G. F. Marchioro, J. M. Daveau, and N.-E. Zergainoh. Multilanguage Specication for System Design and Codesign, chapter 5. Kluwer academic Publishers, 1999.

[55] P. Coste, F. Hessel, A. Jerraya. Multilanguage codesign using SDL and Matlab, 2000.

[56] V. J. Mooney III, G. De Micheli. Real time analysis and priority scheduler generation for hardware-software systems with a synthesized run-time system. In Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design, pages 605-612. IEEE Computer Society, 1997.

[57] S. Schulz and J. Rozenblit. Concepts for model compilation. Proceedings of ICDA Conference, 2000.

[58] M. Varea. Mixed control/data-flow representation for modelling and verification of embedded systems. Technical report, University of Southampton, Mar. 2002.

[59] D. Thomas, J. Adams, and H. Schmit, A Model and Methodology for Hardware-Software Codesign, in IEEEDesign & Test of Computer, vol. 10, pp. 6-15, Sept. 1993.

[60] S.L. Coumeri, D. Thomas. A simulation environment for hardware-software codesign, Proceedings of the International Conference on Computer Design : VLSI in computers & Processors, 1995 Austin, Texas.

[61] E. A. Lee. Overview of the Ptolemy project. Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[62] J. W. Janneck, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, S. Sachs, Y. Xiong. Disciplining heterogeneity : the Ptolemy approach. In ACM SIGPLAN workshop on langages compilers and tools for embedded systems LCTES 2001 Snoowbird Utah, june 2001.

[63] Ptolemy, University of California, Berkeley, USA,http ://ptolemy.eecs.berkeley.edu/ptolemyclassic/body.html.

[64] Modeling distributed hybrid systems in Ptolemy II He Liu ; Xiaojun Liu ; Lee, E.A. ; American Control Conference, Proceedings of 2001, Volume : 6, Page(s) : 4984 -4985.

[65] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, Ptolemy : A Framework for Simulating and Prototyping Heterogeneous Systems, Int. Journal of Computer Simulation, special issue on Simulation Software Development, vol. 4, pp. 155-182, April, 1994.

[66] P. N. Hilfinger. Silage reference manual, draft release 2.0. (1993).

[67] SPW, available on line to www.cadence.com/products/spw.html, 2002.

[68] S. EDWARDS, L. LAVAGNO, E. A. LEE, and A. SANGIOVANNI-VINCENTELLI, "Design of Embedded Systems : Formal Models, Validation, and Synthesis", Proceedings of the IEEE, Vol. 85, No. 3, March 1997.

[69] R. Ernst, J. Henkel, T. Benner. Hardware-software cosynthesis for microcontrollers. Design & Test of Computers, IEEE, Volume : 10 Issue : 4, Page(s) : 64 -75, Dec 1993.

[70] D. Herrmann, J. Henkel, R. Ernst. An approach to the adaptation of estimated cost parameters in the COSYMA system. Proceedings of the Third International Workshop on Hardware/Software Codesign, 22-24 Sep 1994 Page(s) : 100 -107.

[71] AHO, Alfred V., SETHI, Ravi, ULLMAN, Jeffrey D. *Compilateurs* : principes, techniques et outils. s.l. : InterEditions, 1989.

[72] WILSON, R. et al. The SUIF compiler system. Stanford : Stanford University, 1994.

[73] DE MICHELI, G., KU, D., MAILHOT, F., TRUONG, T. The Olympus synthesis system for digital design. IEEE Design & Test of Computers, v.7, n.1, Oct. 1990, p.37-53.

[74] OLYMPUS. Stanford Olympus Synthesis System. Stanford : Stanford University, 1995. http ://akebono.stanford.edu/users/cad/synthesis/olympus.html

[75] R. K. Gupta, G. De Micheli. Specification and analysis of timing constraints for embedded systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume : 16 Issue : 3, Mar 1997, page(s) : 240 -256.

[76] Koo, T.J. ; Sinopoli, B. ; Sangiovanni-Vincentelli, A. ; Sastry, S. A formal approach to reactive system design : unmanned aerial vehicle flight management system design example. International Symposium on , 1999 Computer Aided Control System Design, Proceedings of the 1999 IEEE, Page(s) : 522 -527.

[77] F. Balarin et al. Hardware/Software Co-Design Of Embedded Systems-The Polis Approach. Kluwer Academic Publishers, 2nd printing, 1999.

[78] VIS project Homepage, VIS research group, UC Berkeley, Available on-line at http ://www-cad.eecs.berkeley.edu/Respep/Research/vis/index.html, 2000.

[79] Polis Homepage, Berkeley University, Hardware/software design group, Available on-line at http ://www.eecs.berkeley.edu/ polis, 1999.

[80] Cierto Virtual Component Codesign, Cadence Inc., Available on-line at http ://www.cadence.com/technology/hwsw/ciertovcc, 1999.

[81] N.S. VOROS, L. SANCHEZ, A. ALONSO, A.N. BIRBAS, A.A. JERRAYA, "Hardware/Software Co-design of Complex Embedded Systems - An approach using efficient process models, multiple formalism specication and validation via co-simulation", Design Automation for Embedded Systems, Vol. 8 Issue 1, Kluwer Academic Publishers, March 2003.

[82] C. Valderrama et al. "COSMOS : a transformational codesign tool for multiprocessor architectures in hardware/software codesign", chapter in Hardware/Software Co-design : Principles and Practice (J. Staunstrup and W. Wolf editors), Kluwer, 1997.

[83] T.Ben Ismail, K.O'Brien,A.A.Jerraya, "Interactive system-level partitioning with Partif", Proc. 5th ACM/IEEE Europ. Design and Test Conf., Paris, Paris, France, pp.464-468, Feb. 1994.

[84] Tsasakou, S. Voros, N.S. Koziotis, M. Verkest, D. Prayati, A. Birbas, A. Hardware-software co-design of embedded systems using CoWare's N2C methodology for application development. International Conference on Electronics, Circuits and Systems, 1999. Proceedings of ICECS '99. The 6th IEEE, Volume : 1, Page(s) : 59 -62, 1999.

[85] COWARE. The Coware and Symphony projects. Leuven : IMEC, 1999.
http ://dmz4.imec.be/vsdm/projects/coware_symphony/

[86] Lauwereins, R. ; Engels, M. ; Ade, M. ; Peperstraete, J.A. Grape-II : a system-level prototyping environment for DSP applications Computer, Volume : 28 Issue : 2, Page(s) : 35 -43, Feb 1995.

[87] Lauwereins, R. ; Wauters, P. ; Ade, M. ; Peperstraete, J.A. ; Geometric parallelism and cyclo-static data flow in GRAPE-II Fifth International Workshop on Rapid System Prototyping, 1994.

[88] M. ; Lauwereins, R. ; Peperstraete, J.A. Hardware-software codesign with GRAPE Ade, Rapid System Prototyping, 7-9 Jun 1995 Page(s) : 40 -47.

[89] A. Diagne. Systèmes répartis et coopératifs : une approche multi-formalismes de spécification de systèmes répartis : transformations de composants modulaires en réseaux de petri, Thèse de doctorat, Université Paris 6, 1997.

[90] G. De Micheli, R. Gupta. Hardware/Software Co-design, Proceedings of the IEEE, Vol.85, N.3, pp.349-365, 1997.

[91] M. D. Edwards, J. Forrest, A.E. Whelan. Acceleration of software algorithms using hardware/software co-design techniques. Journal of Systems architecture, Vol. 42, N.9, Feb. 1997.

[92] S.Neema. System-level synthesis of adaptive computing systems, Mar.2000.

[93] A. A. Jerraya, M. Romdhani, P.L. Marrec, F. Hessel, P. Coste, C. Valderama, G. F. Marchiono, J. M. Daveau, et N. -E. Zergainoh. Multilanguage Specification for System Design and Codesign, chapter 5. Kluwer academic Publishers, 1999.

[94] A. M. Turing, On computable numbers, with an application to the Entscheindungs problem, Pro. London Math. Soc., 1936.

[95] F. GECSEG , Products of automata, Spring-Verlag, EATCS Monographs on Theoretical Computer science, 1986.

[96] Z. Lui, C. Corroyer. Effectiveness of heuristics and simulated annealing for the scheduling of concurrent task. an empirical comparison. Proc. of PARLE'93, 5th Int. PARLE conference, Munich, Germany, June 14-17, pages 452-463, Nov. 1993.

[97] Reinaldo A. Bergamaschi, Raul Camposano, Michael Payer. *Data path synthesis using path analysis*. Proceedings of the 28th conference on ACM/IEEE design automation San Francisco, California, United States Pages : 591 - 596, Year of Publication : 1991.

[98] PAPACHRISTOU, Chris, ALZAZERI, Yusuf. A method of distributed controller design for RTL circuits. In : DESIGN, AUTOMATION & TEST IN EUROPE, Munich, 1999.

[99] BENMOHAMMED, Mohamed, KISSION, Polen, RAHMOUNI, Maher, LIEM, Clifford, JERRAYA, Ahmed Amine. Génération automatique de contrôleurs reprogrammables dans un environnement de synthèse de haut niveau. *Technique et science informatique*, v.17, n.10, 1998, p.1277-1297.

[100] BALBONI, A., FORNACIARI, W., SCIUTO, D. Partitioning and exploration strategies in the Tosca co-design flow. In : INTERNATIONAL WORKSHOP ON HARDWARE-SOFWARE CO-DESIGN, 1996.

[101] ANTONIAZZI, S., BALBONI, FORNACIARI, W., SCIUTO, D. A methodology for control-dominated systems codesign. In : INTERNATIONAL WORKSHOP ON HARDWARE-SOFWARE CODESIGN, 1994.

[102] GAJSKI, Daniel D., VAHID, Frank, NARAYAN, Sanjiv, GONG, Jie. System-level exploration with SpecSyn. In : DESIGN AUTOMATION CONFERENCE, 35, San Francisco, 1998.

[103] SpecSyn : an environnement supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v.6, n.1, March 1998, p.84-100.

[104] GUPTA, R.K., DE MICHELI, G. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, v.10, n.3, 1993, p.29-41.

[105] COSYMA. *COSYnthesis for eMbedded Architectures*. ERNST, Rolf (Dir.) Braunschweig : Tecnhical University of Braunschweig, 1998.
http ://www.ida.ing.tu-bs.de

[106] Polen Kission, Hong Ding, and Ahmed A. Jerraya. Structured design methodology for high-level design. In 31st ACM/IEEE Design Automation Conference, 1994.

[107] KISSION, P., CLOSSE, E., BERGHER, L., JERRAYA, A. Industrial experimentation of high-level synthesis environnement. In : EDAC'93, Paris, 1993.

[108] VIEWLOGIC. *FPGA Express*. Viewlogic, 2000.
http ://www.viewlogic.com/ftm/ep_designer.htm

[109] GAJSKI, Daniel D., VAHID, Frank. Specification and design of embedded hardware-software systems. *IEEE Design & Test of Computers*, v.12, n.1, p.53-67, Spring 1995.

[110] PRADO LOPES FILHO, Eudes. *Algorithmes de synthèse de circuits programmables basés sur des graphes de décision binaire*. Paris : Université de Paris VI, 1996. 86p. (Thèse de Doctorat)

[111] MADSEN, J., GRODE, J., KNUDSEN, P. V., PETERSEN, M. E., HAXTHAUSEN. LYCOS : the Lyngby Co-Synthesis-System. *Design Automation for Embedded Systems*, v.2, n.2, 1997.

[112] SYNOPSIS. *Synopsis VHDL compiler reference manual*. s.l. : Synopsis, 1992.

[113] COMPASS. *ASIC synthesizer for VHDL design* : V8R4.0 edition. s.l. : Compass design automation, Nov. 1992.

[114] LEMAÎTRE, Michel, DURRIEU, Guy. Synthèse architecturale par transformation de programmes synchrones purement fonctionnels. *Technique et Science Informatiques*, v.15, n.10, p.1345-1366, 1996.

[115] DURRIEU, G., KESSACI, K., LE MAÎTRE, M. Transe : an experimental transformation assistant for digital circuit design. In : IFIP WG 10.2/WG 10.5 WORKSHOP ON DESIGNING CORRECT CIRCUITS, 2, 1992. *Proceedings*. J. Staunstrup, R. Sharp (Ed.) s.l. : North-Holland, 1992.

[116] LE VERGE, H., MAURAS, C., QUINTON, P. The Alpha language and its use for design of systolic arrays. *Journal of VLSI Signal Processing*, n. 3, p.173-182, 1991.

[117] C. Dezan, H. Le Verge, P. Quinton, and Y. Saouter. The Alpha du Centaur environment. International Workshop Algorithms and Parallel VLSI Architectures, Bonas, France, June 1991.

[118] C. Dezan. Génération automatique de circuits avec ALPHA du CENTAUR. PhD thesis, Université de Rennes 1, Rennes, France, Feb 1993.

[119] D.K. Wilde and O.Sie. Regular array synthesis using alpha. Technical Report 829, IRISA, Rennes, May 1994.

[120] VAN DAGEN, V., PETIT, M. *PRESAGE : a tool for the parallelisation of nested-loop programs*. s.l. : IFIP, North-Holland, 1990. p.341-360.

[121] F. Maraninchi, The Argos language : Graphical Representation of Automata and Description of Reactive Systems, in Proc. IEEE Workshop on Visual Languages, Kobe, Japan, Oct.1991.

[122] E. A. Lee and T. M. Parks. Dataflow process networks. In Proc. of the IEEE, pp. 773-799, May 1995.

[123] N. Halbwachs. The declarative code DC, version 1.2a. Vérimag, Grenoble, France, October 1995. unpublished report, http ://www.inrialpes.fr/bip/people/girault/Documentations/Dcl/index.html.

[124] L.Kaouane, M.Akil, Y.Sorel, T.grandpierre. From algorithm specification to automatic synthesis of FPGA circuit : a seamless flow of graph transformations, FPL03, 13th International Conference on Field Programmable Logic and Applications, Lisbon, Portugal, 1-3 September 2003.

[125] L.Kaouane, M.Akil, Y.Sorel, T.grandpierre. A methodology to implement real-time applications onto reconfigurable circuits, Special issue on Engineering of Configurable Systems of the Journal of Supercomputing, Kluwer Academic Publisher, Vol.30, No.3, Dec. 2004 .

[126] L.Kaouane, M.Akil, Y.Sorel, T.grandpierre. An automated design flow for optimized implementation of real-time image processing applications on FPGA, EUROCON 2003 International Conference on computer as a tool, Ljubljana, Slovenia, 22-24 September 2003.

[127] L.Kaouane, M.Akil, Y.Sorel, T.grandpierre. A methodology to implement real-time applications on reconfigurable circuits, ERSA03, The 2003 International conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA, 23-26 June 2003.

[128] L.Kaouane, M.Akil, Y.Sorel, T.grandpierre. Implantation optimisée sur circuit dédié d'algorithmes spécifiés sous la forme d'un graphe factorisé de dépendances de données : application aux traitements d'images, GRETSI03, 19th symposium on signal and image processing, Paris, 8-11 september 2003.

[129] L.Kaouane, M.Akil, Y.Sorel. Optimized implementation of application specific integrated circuits specified with data dependence graph. EDAA PhD Forum at DATE03, Design Automation and Test in Europe, Munich, Allemagne, 3-7 march 2003.

Annexe A : Construction du graphe temporel

Exemple du produit matrice Vecteur

Nous illustrons le processus de construction du graphe temporel à travers l'exemple complet du produit matrice vecteur.

Niveau hiérarchique 2 : Accumulateur

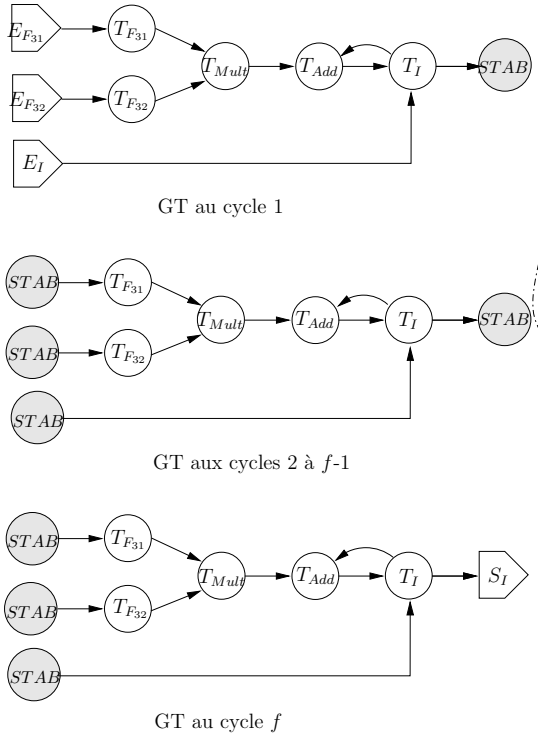


FIG. 1 – Graphes temporels GT_f de la frontière FF_3

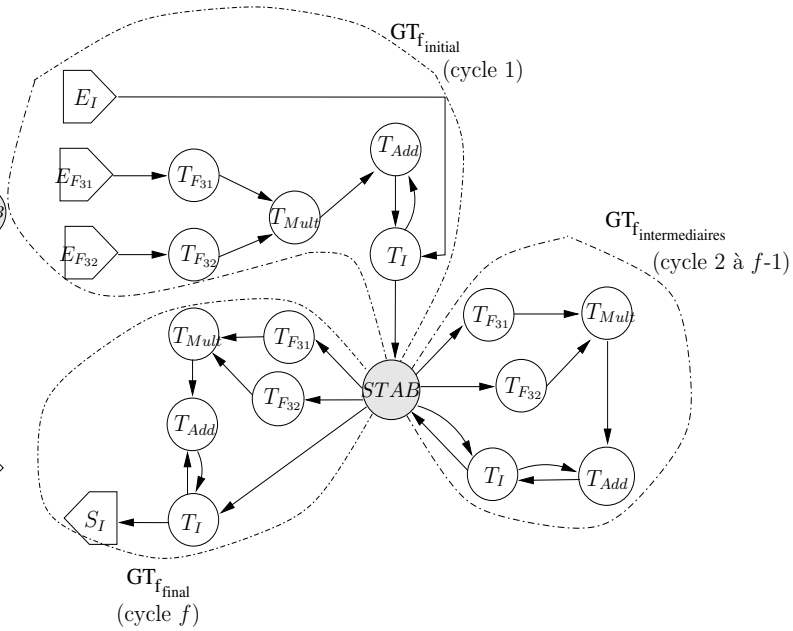


FIG. 2 – Graphe temporel fusionné de la frontière FF_3

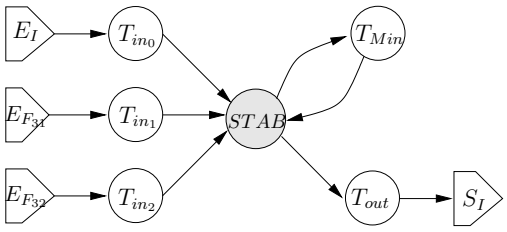


FIG. 3 – Graphe temporel GT_f de la frontière

Où :

- $T_{in_0} = T_{I_{in}} + T_{Add} + T_{I_{out}}$, avec $T_{I_{in}} = 0$ *et* $T_{I_{out}} = T_{setup}$
- $T_{in_1} = T_{F_{31}} + T_{Mult} + T_{Add} + T_I$, avec $T_I = T_{setup}$
- $T_{in_2} = T_{F_{32}} + T_{Mult} + T_{Add} + T_I$, avec $T_I = T_{setup}$
- $T_{Min} = Max(T_{F_{31}} + T_{Mult}, T_{F_{32}} + T_{Mult}, T_{I_{in}}) + T_{Add} + T_{I_{out}}$, avec$T_{I_{in}} = T_{hold}$ $T_{I_{out}} = T_{setup}$
- $T_{out} = Max(T_{F_{31}} + T_{Mult}, T_{F_{32}} + T_{Mult}, T_{I_{in}}) + T_{Add} + T_{I_{out}}$, avec$T_{I_{in}} = T_{hold}$ *et* $T_{I_{out}} = 0$

Niveau hiérarchique 1 : Le Produit Scalaire

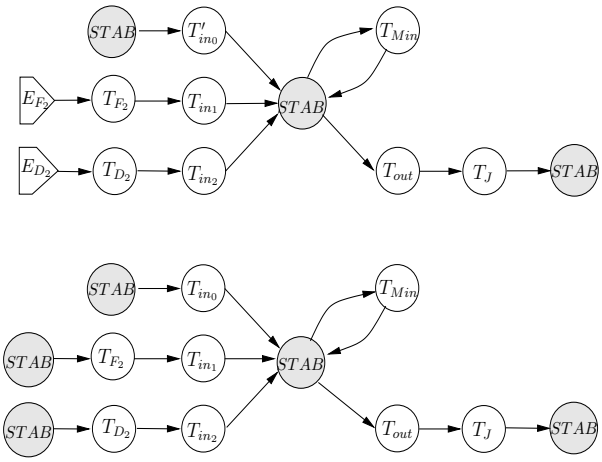


FIG. 4 – Graphes temporels GT_f de la frontière FF_2

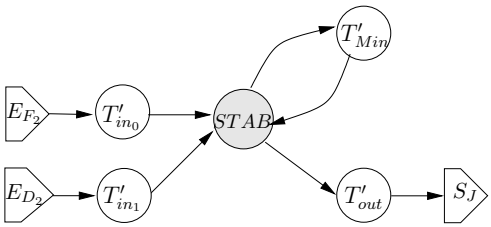


FIG. 5 – Graphe temporel simplifié GT_f de la frontière FF_2

Où :

- $T'_{in_0} = T_{F_2} + T_{in_1}$,
- $T'_{in_1} = T_{D_2} + T_{in_2}$,
- $T'_{out} = Max(T_{out}, T_J)$, *avec* $T_J = T_{hold}$.
- $T'_{Min} = Max \begin{bmatrix} T_{Min} \\ T_{in_0} \\ T_{out} + T_J \\ T_{F_2} + T_{in_1} \\ T_{D_2} + T_{in_2} \end{bmatrix}$ *avec* $T_J = T_{setup}$

Niveau hiérarchique 0 : Accumulateur

On remarque qu'au niveau hiérarchique 0, il n'y ni entrée ni sortie, et que $T"_{Min}$ est en faite le chemin critique T_{crit} global. C'est ce dernier qui détermine la férquence maximale de l'horloge : $F < 1/T_{crit}$.

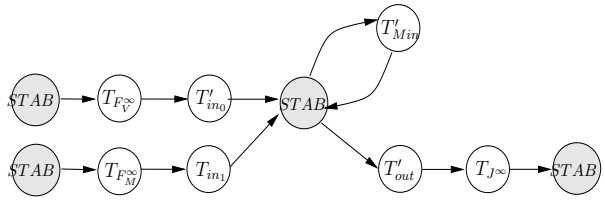


FIG. 6 – Graphe temporel GT_f de la frontière FF_1

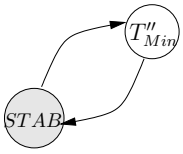


FIG. 7 – Graphe temporel GT_f de la frontière FF_1

Où : $T"_{Min} = Max(T'_{Min}, T_{F_V^\infty} + T'_{in0}, T_{F_M^\infty} + T'_{in1}, T'out + T_J^\infty)$

Annexe B : Caractérisation

Caractérisation des ports de base

Afin de valider notre modèle de caractérisation, nous avons effectué un ensemble de tests sur des applications diverses. Cette validation s'est faite en 3 étapes :

- caractérisation complète des ports (J, F, I...) en vue de la vérification des formules théoriques relatives à chacun de ces ports.
- caractérisation d'opérateurs de base (multiplieurs, additionneurs). Ces valeurs étant relatives à chaque circuit reconfigurable, nous avons choisi comme reférence le FPGA **Virtex 300 BG 432**.
- validation des modules caractérisation et optimisation sur deux exemples : les produit matrice/vecteur et le filtre de Deriche.

Caractérisation des ports de base

Grâce aux logiciels de simulation (Léonardo est utilisé par l'ESIEE), nous avons pu vérifier que les formules théoriques de surface et de temps de traverse pour les ports de base étaient bonnes. Néanmoins, le logiciel utilisé ne faisant que des estimations temporelles, les résultats trouvés ne serviront que de référence, une caractérisation exacte n'étant possible qu'après placement routage. En aucune manière ces chiffres ne pourront donc prétrendre à la réalité.

Ci-dessous, voici des exemples de caractérisation sur des ports de base pour le FPGA **Virtex 300 BG 432**. Ces résultats ont été trouvés sous Léonardo. On les considérera comme reférence.

Résultats sur le port *Iterate*

On utilise des opérateurs sur 8 bits.

- temps de traverse : 4.04
- nombre de FF : 8
- nombre de FG : 8

Les 8 F/G sont dues au mux $2 \rightarrow 1$ sur 8 bits entre la constante d'initialisation et la sortie du registre.

Le chemin critique est dû au temps de chargement du registre + le temps de traversée du mux.

Résultats sur le port *Join*

On utilise des opérateurs sur 8 bits. Attention, on utilise du one hot encoding. On donne d'abord le temps de traversée du port puis le nombre de FF utilisées et enfin le nombre de FG.

Il y a 2 bascules, par CLK. Il faut 7 CLB pour génerer les enables vers les registres (un par registre) Le chemin critique est du au temps de chargement des registres

Join	Caractéristiques		
	temps	nb FF	nb FG
Join	2.39	7	56
Join	2.39	6	48
Join	2.39	5	40
Join	2.39	4	32
Join	2.39	3	24
Join	2.39	2	16

Résultats sur le Multiplexeur

Le multiplexeur étant la brique de base du port *Fork*, nous avons effectué des essais complets sur différents types de multiplexeurs. Nous avons vérifié la formule théorique :

$$2nd \sum_{k=1}^{pr} ceil(f/2^k)$$

Où
- f est le facteur de factorisation. Si on a un multiplexeur $m \rightarrow n$, alors $f = m/n$
- n le nombre de bits sur lequel est codé chaque élément.
- d est le nombre d'éléments entrant.
- $pr = ceil(log_4(f))$

On donne les temps de traverse et le nombre de FG utilisées pour des multiplexeurs $n \rightarrow 1$.

Mux	Caractéristiques		
	temps	nb FF	nb FG
mux $8->1$	3.38	0	5
mux $7->1$	3.38	0	5
mux $6->1$	3.38	0	4
mux $5-1$	3.38	0	3
mux $4-1$	3.38	0	3
mux $3->1$	3.38	0	2
mux $2->1$	1.69	0	1
mux $9->1$	5.07	0	6
mux $15->1$	5.07	0	10
mux $16->1$	5.07	0	11
mux $32->1$	5.07	0	21
mux $33->1$	6.76	0	22
mux $63->1$	6.76	0	42
mux $64->1$	6.76	0	43
mux $90->1$	6.76	0	60
mux $77->1$	6.76	0	52
mux $51->1$	6.76	0	35

Ces valeurs s'accordent parfaitement avec la formule donnée ci-dessus.

Caractérisation d'opérateurs de base

Grâce à la documentation Xilinx, nous avons pu avoir la caractérisation des opérateurs addition et multiplication. On donne le temps de traversée de ces ports, le nombre de FF et de FG utilisées.

- addition : 2950 0 8
- multiplication : 8350 0 64

Annexe C : Bibliothèque VHDL

La bibliothèque VHDL de SynDEx-IC

```
dnl ******************************************************
dnl --Title : LIBRARY VHDL OF COMPONENTS USE IN THIS APPLICATION
dnl --COMPANY : ESIEE
dnl --DEPARTMENT: A2SI

define(‘_Partielpackage’,
type  VECTOR$1 is array (natural RANGE <>) of std_logic_vector($2-1 downto 0);
‘ifelse($#,2,,‘_Partielpackage(shift(shift($@)))’)
’)

define(‘_Package’,
--***************** DEFINITION OF LIBRARYS *****************

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;

--***************** DEFINITION of PACKAGE ******************
--ce module permet de definir les dimmensions du package of application : $1
package definitions is
‘_Partielpackage(shift($@))’
end definitions;
)

define(‘_Title’,----------------------------------------------------------------
-- Title       : AUTOMATIC GENERATION OF VHDL CODE FOR TOP ALGO $1
-- Project     : Automated generation of VHDL with module Gen_vhdl
--------------------------------------------------------------------------------
-- File        : $1_toplevel.vhdl
-- Company     : ESIEE
-- Department  : A2SI
)

define(‘_Gen_lib’,
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--************************************************
```

```
LIBRARY WORK;
USE WORK.definitions.all;
)

define(‘_Gen_entity_uci’,
--***************   DEFINITION OF LIBRARYS  **********
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--**************** DEFINITION DE UC INFINI (entity et architecture) ***************
entity uc_infini is
  port (        rst  : in std_logic;
                en   : out std_logic;
rfd  : out std_logic;
afu  : out std_logic;
rsu  : in std_logic;
asd  : in std_logic;
rfu  : in std_logic;
afd  : in std_logic);
end uc_infini;

architecture OPERATEUR of uc_infini is

Begin
      rfd <= rsu;
      afu <= asd;
      en  <= afd OR rst;

end OPERATEUR;
)

define(‘_Gen_entity_uc’,

--***************   DEFINITION OF LIBRARYS  **********
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--************* DEFINITION DE UC (entity et architecture) ************
entity uc is
generic (nb_bit_compteur :integer:=6 );
port (rsu,afd,rfu,asd, clk,reset    :in std_logic;
      asu,rfd,afu,rsd,en            :out std_logic;
      cpt                           :out std_logic_vector (nb_bit_compteur-1 downto 0));
```

```
end uc;
architecture ucff of uc is
signal compteur: std_logic_vector (nb_bit_compteur-1 downto 0):= (others => '0'); --:="000001";
signal tran    : std_logic:='0';
signal fin     : std_logic;
signal init    : std_logic;
signal enable  : std_logic;
begin

--description de unite de controle
en  <= enable;
fin <= compteur(nb_bit_compteur-1);
rfd <= rsu;
asu <= afd and fin;
cpt <=compteur;
enable  <= (rsu and afd) and ((not fin) or asd);
rsd <= rfu and fin;
afu <= reset or (rsu and (not fin)) or (asd or ((not fin) and (rsu and afd)));
init <= reset or ((afd and fin) and asd);

--description du compteur one hot encoding
process (init, clk, enable)
begin
 if (clk' event and clk = '1') and ( init = '1') then
       compteur(nb_bit_compteur-1 downto 1) <= (others => '0') ;
       compteur(0) <= '1';

 else if (clk' event and clk = '1') and (enable = '1') then
tran <= compteur(nb_bit_compteur-1);
compteur(nb_bit_compteur-1 downto 1) <= compteur (nb_bit_compteur-2 downto 0);
compteur(0) <= tran;
end if;
 end if;
end process;

end ucff;
)

define('_Gen_entity_mseq',
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--***************  DEFINITION  Mseq OPERATION  ******************
entity $1Mseq is
        generic( insize:integer :=4;
               outsize:integer :=12;
 nel:integer :=3;
```

```
 tel:integer :=8);

      port( rst : std_logic;
inport : in VECTOR$1 (insize-1 downto 0) ;
outport : out VECTOR$1 (outsize-1 downto 0) ;
cpt : in std_logic_vector(nel-1 downto 0);
en  : in std_logic;
clk : in std_logic);
end $1Mseq;

architecture OPERATEUR of $1Mseq is

signal  sortie_tmp : VECTOR$1 (outsize-1 downto 0) ;
signal  sortie_tmp2 : VECTOR$1 (insize-1 downto 0) ;

begin

process(sortie_tmp, sortie_tmp2)
begin
for i in 0 to outsize - 2 loop
 outport(i) <= sortie_tmp(i);
end loop;
outport(outsize-1 downto (outsize-insize)) <= sortie_tmp2;
end process;

LAST: process(inport)
begin
sortie_tmp2 <= inport;
end process;

IMPLO : process (clk, en)
begin
       if (clk'event and clk = '1') then
         if (en = '1') then
          for i in 0 to nel - 1 loop
     if (cpt(i) = '1') then
     sortie_tmp(((insize*(i+1)) -1) downto (i*insize)) <= inport(insize-1 downto 0);
    else
    sortie_tmp((insize*(i-1)) downto (i*insize)) <= sortie_tmp((insize*(i-1)) downto (i*insize));
    end if;
    end loop;
     end if;
   end if;
  end process;

end OPERATEUR;
)

define(‘_Gen_entity_xseq’,
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--************************************************
```

```
LIBRARY WORK;
USE WORK.definitions.all;

--****************** DEFINITION Xseq OPERATION  ******************

entity $1oneXseq is

        generic(insize : integer := 8;
              tel : integer := 8);
          port(Entree : in  VECTOR$1 (insize-1 downto 0) ;
            sortie : out std_logic_vector(tel-1 downto 0);
            Commande : in std_logic_vector(insize-1 downto 0)
            );
end $1oneXseq;

architecture OPERATEUR of $1oneXseq is

begin

  IMPLO : process (Entree,Commande)
          variable tmp : VECTOR$1 ( insize-1 downto 0) ;
        variable tmp_ou : std_logic_vector(insize-1 downto 0);

  begin
 for i in 0 to (insize - 1) loop
  for j in 0 to (tel - 1) loop
       tmp(i)(j) := Entree(i)(j) AND Commande(i);
  end loop;
 end loop;
 for i in 0 to (tel - 1) loop
       tmp_ou(0) := tmp(0)(i);
   for j in 1 to (insize - 1) loop
       tmp_ou(j) := tmp_ou(j-1) OR tmp(j)(i);
 end loop;
 Sortie(i) <= tmp_ou(insize-1);
 end loop;
 end process;

end OPERATEUR;

--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--*************************************************

LIBRARY WORK;
USE WORK.definitions.all;

entity $1Xseq is

generic( insize:integer := 12;
         outsize:integer := 4;
```

```
       nel:integer   :=3;
       tel:integer   :=8);
port ( rst : in std_logic;
   inport: in VECTOR$1 (insize -1 downto 0) ;
   outport: out VECTOR$1 (outsize -1 downto 0) ;
   cpt : in std_logic_vector(nel-1 downto 0)
   );
end $1Xseq ;

architecture Xseq_arch of $1Xseq is

type VECTOR_COM is array (natural RANGE <>) of std_logic_vector(insize-1 downto 0);

signal commtemp : std_logic_vector(insize-1 downto 0);
signal tempcom  : VECTOR_COM(0 to outsize-1);

Component $1oneXseq
 generic( insize : integer := 8;
      tel : integer := 8);
 port( Entree : in  VECTOR$1 (insize-1 downto 0);
 sortie : out std_logic_vector(tel-1 downto 0);
 Commande : in std_logic_vector(insize-1 downto 0)
);
End component;

begin
 Xseq1 : for i in 0 to outsize -1 generate
 realXseq: if (insize-outsize>0) generate
  littleXseq:$1oneXseq
  generic map(insize,tel)
  port map (inport, outport(i), tempcom(i));
 end generate realXseq;
 diffuse: if (insize-outsize=0) generate
outport(i) <= inport(i);
 end generate diffuse;
end generate Xseq1;
sigcom : process(cpt,tempcom,commtemp)
variable iter : integer :=((insize-outsize)/(nel-1));
begin
     if cpt'event then
      for i in 0 to nel-1 loop
     if cpt(i)='1' then
     tempcom(0)<=('0',others=>'0');
     tempcom(0)((iter*i))<='1';
    end if;
    end loop;
   end if;
   for i in 1 to outsize-1 loop
tempcom(i)<=tempcom(i-1)(insize-2 downto 0) & '0';
   end loop;
end process sigcom;

end Xseq_arch;

)

define(‘_Gen_entity_iseq’,
```

```
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--********************* DEFINITION Iseq OPERATION ******************

entity $1Iterate is
  generic ( size  : integer :=1;
            nel : integer :=3;
         tel : integer :=8
        );
  port (rst : in std_logic;
    I_init : in VECTOR$1 (size-1 downto 0) ;
    inport  : in VECTOR$1 (size-1 downto 0) ;
    O_si  : out VECTOR$1 (size-1 downto 0) ;
    outport  : out VECTOR$1 (size-1 downto 0) ;
    cpt: in std_logic_vector(nel-1 downto 0);
    en  : in std_logic;
    clk : in std_logic
     );
end $1Iterate;

architecture OPERATEUR of $1Iterate is
  signal Retard  : VECTOR$1 (size -1  downto 0) ;
  begin
      I : process (clk)
          begin
        if (clk'event and (clk = '1')) then
        if (en = '1') then
          Retard <= inport;
      end if;
      end if;
end process I;

  process(cpt)
     begin
         if (cpt(0) = '0') then
            O_si <= Retard;
      else
         O_si <= I_init;
      end if;
     end process;

     outport <= inport;
end OPERATEUR;

)

define(`_Const_entity_core',`ifelse($#,3,,
```

```
‘$1(incr($2)) <= conv_std_logic_vector ($4,tel1);
_Const_entity_core($1,incr($2),shift(shift(shift($@))))’)
’)

define(‘_Gen_entity_const’,
‘ifelse($#,incr(incr(incr(incr(incr($4))))),

--********************** DEFINITION OF LIBRARYS ******************
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--*****************************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--************  DEFINITION CONSTANTE OPERATION **********************

entity $1$2 is
       generic( size1 : integer :=$4;
                tel1 : integer :=$5
            );
      port( $3 : out VECTOR$6 (size1-1 downto 0)
        );
end $1$2;

architecture OPERATEUR of $1$2 is
begin
$3(0) <= conv_std_logic_vector ($7,tel1);
_Const_entity_core($3,0,shift(shift(shift(shift(shift(shift($@)))))))
end OPERATEUR;, pierre--COMPONENT $1$2 NOT DEFINED; COMPILE IT SEPARATLY
_Gen_entity_const_not_in_lib($@) )
’)

define(‘_Gen_entity_memory’,
‘ifelse($#,incr(incr(incr(incr(incr(incr(incr(incr(incr(incr($5)))))))))),
--********************** DEFINITION OF MEMORY ***********************
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--*****************************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--********************  DEFINITION OF MEMORY OPERATION ***************
entity $1$2 is
generic(size1 : integer := $5;
tel1 : integer := $6;
size2 : integer := $10;
tel2 : integer := $11
);
port(
```

```
$3 : $4 VECTOR$7(size1 -1 downto 0);
$8 : $9 VECTOR$12(size2 -1 downto 0);
en : in std_logic;
clk : in std_logic;
rst : in std_logic
    );
end $1$2;

architecture OPERATEUR of $1$2 is
signal tabvect : VECTOR$7(size2-1 downto 0) ;
begin

$8 <= tabvect ;
process(clk,rst,en)
  begin
  if rst = ''1'' then
 tabvect(0) <= conv_std_logic_vector ($13,tel1);
_Const_entity_core(tabvect,0,shift(shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift($@))))))))))))
  elsif '(clk'event and clk = '1') then
    if en = '1'' then
   tabvect <= $3;
  end if;
  end if;
end process;
end OPERATEUR;, --COMPONENT $1$2 NOT DEFINED; COMPILE IT SEPARATLY
_Gen_entity_memory_not_in_lib($@))

')

define('_Gen_entity_actu_partiel',
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--*************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--*********************  DEFINITION ACTUATOR OPERATION  **********************

entity $1$2 is
        generic( size1 : integer :=$4;
               tel1 : integer :=$5
      );
port (
$3  : in VECTOR$6(size1-1 downto 0);
en  : in std_logic;
clk : in std_logic;
rst : in std_logic
   );
end $1$2;
```

```
architecture OPERATEUR of $1$2 is

type tableau is array (9 downto 0) of VECTOR$6 (size1-1 downto 0);
signal tabvect : tableau;
signal number : integer := 0;
begin
PRO : process (rst, clk, en)
   begin
      if rst = '1' then
           for j in 0 to 9 loop
       for i in 0 to (size1-1) loop
             tabvect(j) (i) <= (others => '0');
       end loop;
     end loop;
    elsif (clk'event) and (clk = '1') then
      if en = '1' then
       if  number < 10 then
    tabvect(number) <= i;
    number <= number +1;
     else
    number <= 0;
     end if;
      end if;
       end if;
end process;

end OPERATEUR;
)

define(`_Gen_entity_sens',`_Gen_entity_sens_not_in_lib($@)')

define(`_Gen_entity_actu',
`ifelse($#,6, _Gen_entity_actu_partiel($@),--COMPONENT $1$2 NOT DEFINED; COMPILE IT SEPARATLY
_Gen_entity_actu_not_in_lib($@))
')

define(`_Gen_entity_diffuse',
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--************** DEFINITION DIFFUSE OPERATION ******************

entity $1Diffuse is
      generic(       size:integer :=4;
                      tel:integer :=8);
      port (
          rst : std_logic;
        inport : in VECTOR$1 (size-1 downto 0) ;
```

```
         outport : out VECTOR$1 (size-1 downto 0) );
end $1Diffuse;

architecture OPERATEUR of $1Diffuse is
begin
        outport <= inport;
end OPERATEUR;

)

define(‘_Gen_generic_cin’, ‘ifelse($#, 7,size$1 : integer := $4;
           tel$1  : integer := $5,
       size$1 : integer := $4;
       tel$1  : integer := $5;
      ‘_Gen_generic_cin(incr($1),shift(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Gen_port_cin’,‘ifelse($#, 7,$2 : out VECTOR$6 (size$1 -1 downto 0);
       rd$3 : out std_logic;
      ad$3 : in  std_logic, $2 : out VECTOR$6 (size$1 -1 downto 0);
    rd$3 : out std_logic;
   ad$3 : in  std_logic;
 ‘_Gen_port_cin(incr($1),shift(shift(shift(shift(shift(shift(shift($@)))))))’)
’)
define(‘_Signal_archi_cin’, signal sig$1 : VECTOR$5($3-1 downto 0);
   signal flag$6 : std_logic := ’1’;
  ‘ifelse($#, 6,,‘_Signal_archi_cin(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Signal_archi_cout’, signal flag$6 : std_logic :=’0’;
‘ifelse($#, 6,,‘_Signal_archi_cout(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Def_output_cin’, $1 <= sig$1;
     rd$2 <= ru and flag$6 ;
 ‘ifelse($#, 6,,‘_Def_output_cin(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Def_output_cout’, au$2 <= ad and flag$6;
 ‘ifelse($#, 6,,‘_Def_output_cout(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Def_au_cin’, ‘ifelse($#,6,, or ad$8 ‘_Def_au_cin(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Def_rd_cout’, ‘ifelse($#,6,, or ru$8 ‘_Def_rd_cout(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

dnl define(‘_Def_proc_cout’, ‘ifelse($#,6,, ‘, ru$8 _Def_proc_cout(shift(shift(shift(shift(shift(shift($@))))
dnl’)

dnl define(‘_Init_cin’, for i in 0 to ($3-1) loop
dnl sig$1(i) <= (others => ’0’);
dnl     end loop;
dnl     ‘ifelse($#, 6,,‘_Init_cin(shift(shift(shift(shift(shift(shift($@)))))))’)
dnl’)
```

```
define(‘_Fct_intermediaire’, ’0’ ‘when $1,’)

dnl define(‘_Decr_for_flag’, ‘ifelse($1,$2,,‘_Fct_intermediaire($1)’
dnl ‘_Decr_for_flag(incr($1),$2)’)
dnl ’)

dnl define(‘_Down_flag’, ‘ifelse($#, 6,,‘_Fct_intermediaire($12)’
dnl     ‘_Down_flag(shift(shift(shift(shift(shift(shift($@)))))))’)
dnl ’)
dnl define(‘_Process_archi_cin’, elsif $1(0) = $8 then  sig$3 <= $2; flag$8 <= ’1’; flagnotsymetric <= ’0’; ‘
dnl ‘ifelse($#, 8,else ‘_Decr_for_flag(0,incr($8))’,‘_Process_archi_cin($1,$2,shift(shift(shift(shift(shift(s
dnl’)

define(‘_Select_archi_cin_port’, ‘with conv_integer($1(0)) select
sig$3 <= $2 when $8,
sig$3 when others;
     ifelse($#,8,,‘_Select_archi_cin_port($1,$2,shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))’)
’)

define(‘_Select_archi_cout_port’, ‘$1 when $6,
ifelse($#,6,,‘_Select_archi_cout_port(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Select_archi_cin_flag’, ‘with conv_integer($1(0)) select
                                        flag$7 <= ’’1’’ when $7,
          ’’0’’ when others;
ifelse($#,7,,‘_Select_archi_cin_flag($1,shift(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Select_archi_cin_flagns’, ‘ifelse($#,6,,‘_Fct_intermediaire($12)’
‘_Select_archi_cin_flagns(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Gen_generic_opn’, ‘ifelse($#, 6,size$1 : integer := $4;
        tel$1  : integer := $5,
    size$1 : integer := $4;
        tel$1  : integer := $5;
     ‘_Gen_generic_opn(incr($1),shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Gen_port_opn’, ‘$2 : $3 VECTOR$6 (size$1 -1 downto 0) ;
ifelse($#, 6,,‘_Gen_port_opn(incr($1),shift(shift(shift(shift(shift(shift($@)))))))’)
’)

dnl define(‘_Process_archi_cout’, elsif ru$3 = ’1’  then  sig$1 <= $2; flag$7 <= ’1’; ‘_Decr_for_flag(0,$7)’
dnl ‘ifelse($#, 7,,‘_Process_archi_cout($1,shift(shift(shift(shift(shift(shift(shift($@)))))))’)
dnl’)

define(‘_Gen_archi_xpar’, ‘ifelse($#, 7,
$3 <= $1 ( $2 -1 downto eval($2 - $5));,
     $3 <= $1 ( $2 -1 downto eval($2 - $5));
    ‘_Gen_archi_xpar($1,eval($2 - $5),shift(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Gen_archi_mpar’, ‘ifelse($#, 7,
$1 ( eval($2 + $5) -1 downto $2 ) <= $3;,
    $1 ( eval($2 + $5) -1 downto $2 ) <= $3;
```

```
    `_Gen_archi_mpar($1,eval($2 + $5),shift(shift(shift(shift(shift(shift(shift($@)))))))')
')

define(`_Gen_entity_mpar',
--********************** DEFINITION OF LIBRARIES *****************

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;

--*********************************************************************
LIBRARY WORK;
USE WORK.definitions.all;

--******************** DEFINITION OPERATION MPAR *******************
entity $2 is
GENERIC ( size1 : integer := $5;
    tel1  : integer := $6;
    size2 : integer := $10;
    tel2  : integer := $11;
`_Gen_generic_opn(3,shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift($@))))))))))))'
);

PORT (
`_Gen_port_opn(1,shift(shift($@)))'
             rst : in std_logic
);
end $2;

architecture OPERATEUR of $2 is
begin
$3 (size2 -1 downto 0) <= $8;
`_Gen_archi_mpar($3, $10,shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift(shift($@))))))))))))'
end OPERATEUR;

)

define(`_Gen_entity_xpar',
--********************** DEFINITION OF LIBRARYS ***********************

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--*************************************************************************
LIBRARY WORK;
USE WORK.definitions.all;

--********************** DEFINITION OPERATION XPAR ***********************

entity $2 is
GENERIC ( size1 : integer := $5;
```

```
    tel1  : integer := $6;
    size2 : integer := $10;
    tel2  : integer := $11;
‘_Gen_generic_opn(3,shift(shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift($@))))))))))))’
 );

PORT (
‘_Gen_port_opn(1,shift(shift($@)))’
    rst : in std_logic
   );
end $2;

architecture OPERATEUR of $2 is
begin
$8 <= $3 (size1 -1 downto ‘eval($5 - $10)’);
‘_Gen_archi_xpar($3, eval($5 - $10),shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift(shift($@))))))))))))’

end OPERATEUR;
)

define(‘_Gen_entity_cin’,
--********************** DEFINITION OF LIBRARYS ************************

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--*************************************************************************
LIBRARY WORK;
USE WORK.definitions.all;

--******************* DEFINITION CONDITION OPERATION IN ********************

entity $1 is
GENERIC( size1 : integer := $3;
     tel1  : integer := $4;
     size2 : integer := $7;
     tel2  : integer := $8;
     ‘_Gen_generic_cin(3,shift(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))’
                       );
PORT (rst : in std_logic;
    $2  : in VECTOR$5 (size1 -1 downto 0);
    $6  : in VECTOR$9 (size2 -1 downto 0);
    ru  : in std_logic;
    au  : out std_logic;
    ‘_Gen_port_cin(3,shift(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))’
   );
end $1;

architecture OPERATEUR of $1 is

signal flagnotsymetric : std_logic := ’0’;
‘_Signal_archi_cin(shift(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))’
begin
```

```
au <= flagnotsymetric or ad$11 '_Def_au_cin(shift(shift(shift(shift(shift(shift
(shift(shift(shift($@)))))))))');
'_Def_output_cin(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))'
dnl '_Init_cin(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))'
'_Select_archi_cin_port($2,$6,shift(shift(shift(shift(shift(shift(shift
(shift(shift($@)))))))))'
'_Select_archi_cin_flag($2,shift(shift(shift(shift(shift(shift(shift(shift
(shift($@)))))))))'
'with conv_integer($2(0)) select
 flagnotsymetric <= ''0'' when $15,
_Select_archi_cin_flagns(shift(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))
''1'' when others;

end OPERATEUR;
')

define('_Gen_port_cout','ifelse($#, 7,$2 : in VECTOR$6 (size$1 -1 downto 0);
       ru$3 : in std_logic;
    au$3 : out std_logic, $2 : in VECTOR$6 (size$1 -1 downto 0);
    ru$3 : in std_logic;
 au$3 : out std_logic;
  '_Gen_port_cout(incr($1),shift(shift(shift(shift(shift(shift(shift($@)))))))')
')

define('_Gen_entity_cout',
--************************** DEFINITION OF LIBRARYS ******************

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
USE IEEE.std_logic_arith.all;
--*********************************************************************
LIBRARY WORK;
USE WORK.definitions.all;

--*************** DEFINITION CONDITION OPERATION OUT ******************

entity $1 is
GENERIC (
size1 : integer := $3;
tel1  : integer := $4;
size2 : integer := $7;
tel2  : integer := $8;
'_Gen_generic_cin(3,shift(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))'
);
PORT (
rst : in std_logic;
$2  : in VECTOR$5 (size1 -1 downto 0);
$6  : out VECTOR$9 (size2 -1 downto 0);
rd  : out std_logic;
ad  : in std_logic;
'_Gen_port_cout(3,shift(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))'
);
end $1;

architecture OPERATEUR of $1 is
```

```
signal sig$6 : VECTOR$9($7-1 downto 0);
'_Signal_archi_cout(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))'
begin
$6 <= sig$6;
rd <= ru$11 '_Def_rd_cout(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))';
'_Def_output_cout(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))'
with conv_integer($2(0)) select
sig$6 <=
'_Select_archi_cout_port(shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))'
sig$6 when others;
'_Select_archi_cin_flag($2,shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))'
end OPERATEUR;

)

define('_Gen_entity_mul',
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
--************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--********************  DEFINITION MULTIPLICATION OPERATION ******************
entity $1Arit_mul is
   generic ( size1 : integer := $5;
               tel1  : integer := $6;
         size2 : integer := $10;
          tel2  : integer := $11;
         size3 : integer := $15;
         tel3 : integer := $16
         );
  port ( rst : in std_logic;
     b : in VECTOR$7(size1-1 downto 0);
     a : in VECTOR$12(size2-1 downto 0);
     o : out VECTOR$17(size3-1 downto 0)
     );
end $1Arit_mul;

architecture OPERATEUR of $1Arit_mul is
signal sortiemul : signed(2*tel1-1 downto 0);

begin
  sortiemul <= signed(a(0)) * signed(b(0));
    o(0) <= std_logic_vector(sortiemul(tel3-1 downto 0));
end OPERATEUR;
)

define('_Gen_entity_add',
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
```

```
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
--*************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--***********************   DEFINITION ADDITION OPERATION   *********************
entity $1Arit_add is
   generic( size1 : integer := $5;
            tel1  : integer := $6;
          size2 : integer := $10;
       tel2  : integer := $11;
       size3 : integer := $15;
       tel3 : integer := $16
       );
  port (
      rst : in std_logic;
    b : in VECTOR$7 (size1 -1 downto 0) ;
    a : in VECTOR$12 (size2 -1 downto 0) ;
    o : out VECTOR$17 (size3 -1 downto 0)
     );
end $1Arit_add;

architecture OPERATEUR of $1Arit_add is
signal sortieadd  : signed(tel3-1 downto 0);
begin
       sortieadd <= signed(a(0)) + signed(b(0));
     o(0) <= std_logic_vector(sortieadd);

end OPERATEUR;

)

define('_Gen_entity_sub',
--***************   DEFINITION OF LIBRARYS  **********

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
--*************************************************

LIBRARY WORK;
USE WORK.definitions.all;
--********************  DEFINITION SUBSTRACTION OPERATION  *****************

entity $1Arit_sub is
   generic(size1 : integer := $5;
            tel1  : integer := $6;
          size2 : integer := $10;
       tel2  : integer := $11;
       size3 : integer := $15;
       tel3  : integer := $16
        );
  port(
   rst : in std_logic;
```

```
   b : in VECTOR$7 (size1 -1 downto 0) ;
   a : in VECTOR$12 (size2 -1 downto 0) ;
   o : out VECTOR$17 (size3 -1 downto 0)
    );
end $1Arit_sub;

architecture OPERATEUR of $1Arit_sub is
signal sortiesub : signed(tel3-1 downto 0);
begin
        sortiesub <= signed(b(0)) - signed(a(0));
        o(0) <= std_logic_vector(sortiesub);

end OPERATEUR;

)

define(‘_Gen_entity_window’,
--*****************************  DEFINITION OF LIBRARYS   *****************
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
USE IEEE.numeric_std.all;
--**************************************************

LIBRARY WORK;
USE WORK.definitions.all;

--**************************  DEFINITION WINDOW OPERATION *****************

entity $1window is
generic(size1 : integer := $5;
        tel1 : integer := $6;
    size2 : integer := $10;
    tel2 : integer := $11);

port (clk : in std_logic;
  en  : in std_logic;
  i   : in VECTOR$7(size1-1 downto 0);
  o   : out VECTOR$12(size2-1 downto 0);
  rst : in std_logic);
 end $1window;

architecture OPERATEUR of floatwindow is

signal tabvect : VECTORfloat(size2-1 downto 0) ;
begin
 o <= tabvect ;

process(rst,clk,en)
begin
 if rst = ’1’ then
  for i in 0 to (size2-1) loop
   tabvect(i) <= (others => ’0’);
  end loop;
 elsif (clk’event) and (clk = ’1’) then
  if en = ’1’ then
      tabvect(size2-2 downto 0) <= tabvect(size2-1 downto 1);
```

```
      tabvect(size2-1) <= i(0);
  end if;
end if;
end process;

end OPERATEUR;
)

define(‘_Gen_entity_opn’,
‘
ifelse($2,Arit_mul,‘_Gen_entity_mul($@)’,
ifelse($2,Arit_add,‘_Gen_entity_add($@)’,
ifelse($2,Arit_sub,‘_Gen_entity_sub($@)’,
ifelse($2,window,‘_Gen_entity_window($@)’,‘_Gen_entity_opn_not_in_lib($@)’))))

’)

define(‘_Gen_entity’,
--***************   DEFINITION OF ENTITY   **********************
ENTITY $1 IS
PORT (
clk : in std_logic;
rst : in std_logic
     );

END $1;
)

define(‘_Gen_comp_uci’,
‘--***************   COMPONENT DEFINITIONS   *********************
--*************** COMPONENT uc_infini *********************
ARCHITECTURE $1 OF $2 is
COMPONENT uc_infini
PORT (
rst : in std_logic;
en  : out std_logic;
rsu : in std_logic;
asd : in std_logic;
rfu : in std_logic;
afu : out std_logic;
rfd : out std_logic;
afd : in std_logic
);

END COMPONENT;

’)

define(‘_Gen_comp_uc’,
‘--***************   COMPONENT uc   *********************
COMPONENT uc
GENERIC ( nb_bit_compteur  : integer := 3);

  PORT ( reset  : in std_logic;
         en   : out std_logic;
       clk  : in std_logic;
         rsu  : in std_logic;
```

```
       asu  : out std_logic;
      rsd  : out std_logic;
         asd  : in std_logic;
         rfu  : in std_logic;
         afu  : out std_logic;
      rfd  : out std_logic;
         afd  : in std_logic;
      cpt  : out std_logic_vector(nb_bit_compteur-1 downto 0)
  );
END COMPONENT;
')

define(‘_Gen_comp_xseq’,
‘--***************** COMPONENT Xseq    ******************
COMPONENT $1Xseq
GENERIC ( insize  : integer := 3;
      outsize : integer := 1;
      nel : integer := 1;
      tel : integer := 8
       );
PORT (
   rst : in std_logic;
  inport  : in VECTOR$1 (insize-1 downto 0) ;
  outport : out VECTOR$1 (outsize-1 downto 0);
     cpt : in std_logic_vector(nel-1 downto 0)
     );
END COMPONENT;
')

define(‘_Gen_comp_mseq’,
‘--***************  COMPONENT Mseq   **********************
COMPONENT $1Mseq
GENERIC ( insize  : integer := 1;
   outsize : integer := 3;
   nel : integer := 1;
      tel : integer := 8
   );
  PORT (
   rst    : in std_logic;
inport : in VECTOR$1 (insize-1 downto 0) ;
clk : in std_logic;
outport : out VECTOR$1 (outsize-1 downto 0) ;
  cpt : in std_logic_vector(nel-1 downto 0);
en  : in std_logic
);
END COMPONENT;
')

define(‘_Gen_comp_iterate’,
‘--************************  COMPONENT ITERATE ******************
COMPONENT  $1Iterate
GENERIC ( size : integer :=1;
   nel  : integer :=3;
tel  : integer :=8
);
PORT (
   rst    : in std_logic;
```

```
I_init : in VECTOR$1 (size-1 downto 0) ;
inport : in VECTOR$1 (size-1 downto 0) ;
clk    : in std_logic;
O_si   : out VECTOR$1 (size-1 downto 0) ;
outport : out VECTOR$1 (size-1 downto 0) ;
  cpt    : in std_logic_vector(nel-1 downto 0);
en     : in std_logic
);
END COMPONENT;
')

define(‘_Gen_comp_diffuse’,
‘--**********************   COMPONENT DIFFUSE ***************
COMPONENT $1Diffuse
GENERIC ( size : integer :=1;
tel  : integer :=8
);
PORT (
rst : in std_logic;
inport  : in VECTOR$1 (size-1 downto 0) ;
outport : out VECTOR$1 (size-1 downto 0)
);
END COMPONENT;

')

define(‘_Gen_comp_cin’,
‘--****************************  COMPONENT CONDITION IN *************
COMPONENT $1
GENERIC (
size1 : integer := $3;
tel1  : integer := $4;
size2 : integer := $7;
tel2  : integer := $8;
_Gen_generic_cin(3,shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))
);
PORT (
rst : in std_logic;
$2  : in VECTOR$5 (size1 -1 downto 0);
$6  : in VECTOR$9 (size2 -1 downto 0);
ru  : in std_logic;
au  : out std_logic;
_Gen_port_cin(3,shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))
);
END COMPONENT;
')

define(‘_Gen_port_cout’,‘ifelse($#, 7,$2 : in VECTOR$6 (size$1 -1 downto 0);
      ru$3 : in std_logic;
      au$3 : out std_logic, $2 : in VECTOR$6 (size$1 -1 downto 0);
    ru$3 : in std_logic;
    au$3 : out std_logic;
       ‘_Gen_port_cout(incr($1),shift(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Gen_comp_cout’,
‘--****************************  COMPONENT CONDITION OUT *************
```

```
COMPONENT $1
GENERIC (
size1 : integer := $3;
tel1  : integer := $4;
size2 : integer := $7;
tel2  : integer := $8;
_Gen_generic_cin(3,shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))
);
PORT (
rst : in std_logic;
$2  : in VECTOR$5 (size1 -1 downto 0);
$6  : out VECTOR$9 (size2 -1 downto 0);
rd  : out std_logic;
ad  : in std_logic;
_Gen_port_cout(3,shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))
);
END COMPONENT;
')

define(‘_Gen_generic’, ‘ifelse($#, 5, size$1 : integer := $3;
 tel$1  : integer := $4,
 size$1 : integer := $3;
 tel$1  : integer := $4;
‘_Gen_generic(incr($1),shift(shift(shift(shift(shift($@))))))’)
’)

define(‘_Gen_port_s’,
  ‘ifelse($#, 5, $2 : out VECTOR$5 (size$1 -1 downto 0) , $2 : out VECTOR$5 (size$1 -1 downto 0) ;
‘_Gen_port_s(incr($1),shift(shift(shift(shift(shift($@))))))’)
’)

define(‘_Gen_comp_sens’,
‘--************** COMPONENT SENSOR  *********************
COMPONENT $1$2
GENERIC (
     _Gen_generic(1,shift(shift($@)))
        );
PORT (
   rst : in std_logic;
   clk    : in std_logic;
  en     : in std_logic;
  _Gen_port_s(1,shift(shift($@)))
     );
END COMPONENT;
’)

define(‘_Gen_comp_const’,
‘--************** COMPONENT CONSTANTE  *********************
COMPONENT $1$2
GENERIC (
_Gen_generic(1,shift(shift($@)))
);
PORT (
_Gen_port_s(1,shift(shift($@)))
  );
END COMPONENT;
’)
```

```
define(‘_Gen_port_a’, ‘$2 : in VECTOR$5 (size$1 -1 downto 0) ;
ifelse($#, 5,,‘_Gen_port_a(incr($1),shift(shift(shift(shift(shift($@))))))’)
’)
define(‘_Gen_comp_actu’,
‘--************************   COMPONENT ACTUATOR    ****************
COMPONENT $1$2
GENERIC (
_Gen_generic(1,shift(shift($@)))
  );
PORT (
rst : in std_logic;
clk : in std_logic;
_Gen_port_a(1,shift(shift($@)))
en  : in std_logic
    );
END COMPONENT;
’)

define(‘_Gen_generic_opn’, ‘ifelse($#, 6,size$1 : integer := $4;
  tel$1  : integer := $5,
size$1 : integer := $4;
  tel$1  : integer := $5;
‘_Gen_generic_opn(incr($1),shift(shift(shift(shift(shift(shift($@)))))))’)
’)
define(‘_Gen_port_opn’, ‘$2 : $3 VECTOR$6 (size$1 -1 downto 0) ;
 ifelse($#, 6,,‘_Gen_port_opn(incr($1),shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Gen_comp_opn’,
‘ifelse($2,window,
--************************  COMPONENT WINDOW **********************
COMPONENT $1$2
GENERIC (
_Gen_generic_opn(1, shift(shift($@)))
);
PORT (
_Gen_port_opn(1,shift(shift($@)))
rst : in std_logic;
clk : in std_logic;
en  : in std_logic
    );
END COMPONENT;,
--************************    COMPONENT OPERATOR   **********************
COMPONENT $1$2
GENERIC (
_Gen_generic_opn(1, shift(shift($@)))
);
PORT (
_Gen_port_opn(1,shift(shift($@)))
rst : in std_logic
    );
END COMPONENT;)
’)

define(‘_Gen_comp_xpar’,
‘--************************** COMPONENT XPAR OR MPAR  ********************
```

```
COMPONENT $2
GENERIC (
_Gen_generic_opn(1, shift(shift($@)))
);
PORT (
_Gen_port_opn(1,shift(shift($@)))
rst : in std_logic
);
END COMPONENT;
')

define(‘_Gen_comp_mpar’, ‘_Gen_comp_xpar($@)’
)

define(‘_Gen_comp_memory’,
‘--****************************** COMPONENT MEMORY  ********************
COMPONENT $1$2
GENERIC (
_Gen_generic_opn(1, shift(shift($@)))
);
PORT (
_Gen_port_opn(1,shift(shift($@)))
rst : in std_logic;
clk : in std_logic;
en : in std_logic
);
END COMPONENT;
')

define(‘_Sig_array’,‘signal $1 : VECTOR$4 ( $2 -1 downto 0); ’)
define(‘_Sig_std’, ‘signal $1 : std_logic;’)
define(‘_Sig_vect’,‘signal $1 : std_logic_vector ($2 -1 downto 0);’)
define(‘_Cnx_uc_inf_combi’,
signal highstate : std_logic :=’1’;
signal lowstate : std_logic := ’0’;
‘--**********  CONNEXION UC INFINITE FOR APPLICATION ONLY COMBINATOIRE  *********
 -- Liste des operations controlees par $1 uc  : $2
 -- Liste des uc situees du cote fast en amont de $1 uc : $3
 -- Liste des uc situees du cote fast en aval de $1 uc  : $4
 begin
  $1te : uc_infini
PORT MAP (
rst => rst,
en  => enableinf,
rsu => highstate,
asd => highstate,
rfu => lowstate,
afu => afuinfini,
rfd => rfdinfini,
afd => highstate
);
')

define(‘_Cnx_uc_inf’,
signal highstate : std_logic :=’1’;
‘--************  CONNEXION UC INFINITE FOR APPLICATION WITH SEQUENTIEL OPERATORS *******
 -- Liste des operations controlees par $1 uc  : $2
```

```
 -- Liste des uc situees du cote fast en amont de $1 uc : $3
 -- Liste des uc situees du cote fast en aval de $1 uc  : $4
 begin
  $1te : uc_infini
PORT MAP (
rst => rst,
en  => enableinf,
rsu => highstate,
asd => highstate,
rfu => $5,
afu => afuinfini,
rfd => rfdinfini,
afd => $7
);
ifelse($5,$6,, $5 <= $6;)
ifelse($7,$8,, $7 <= $8;)
')

define(`_Cnx_uc',
`--*********  CONNEXION UC FINITE FOR APPLICATION WITH SEQUENTIEL OPERATORS  **********
 -- Liste des operations controlees par $1 uc  : $2
 -- Liste des uc situees du cote fast en amont de $1 uc : $3
 -- Liste des uc situees du cote fast en aval de $1 uc  : $4

  $1 : uc
GENERIC MAP ($5)
PORT MAP (
reset => rst,
en  => enable$6,
clk => clk,
rsu => $7,
asu => asu$6,
rsd => rsd$6,
asd => $9,
rfu => $11,
afu => afu$6,
rfd => rfd$6,
afd => $13,
cpt => compteur$6
);
ifelse($7,$8,, $7 <= $8;)
ifelse($9,$10,, $9 <= $10;)
ifelse($11,$12,, $11<= $12;)
ifelse($13,$14,, $13<= $14;)
')


define(`_Cnx_xseq',
`--****************** CONNEXION OF Xseq OPERATOR ******************
 -- La reference de loperation Xseq : $2
 -- Le chemin de loepation Xseq : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

  $2 : $1Xseq
GENERIC MAP ($5, $6, $7, $8)
PORT MAP (
rst     => rst,
```

```
inport  => $9,
outport => $10,
cpt     => compteur$4
);
')

define(‘_Cnx_mseq’,
‘--******************** CONNEXION OF Mseq OPERATOR *******************
 -- La reference de loperation Mseq : $2
 -- Le chemin de loepation Mseq : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

  $2 : $1Mseq
GENERIC MAP ($5, $6, $7, $8)
PORT MAP (
rst     => rst,
inport  => $9,
outport => $10,
cpt     => compteur$4,
clk     => clk,
en      => enable$4
);
')

define(‘_Cnx_iterate’,
‘--******************** CONNEXION OF Iterate OPERATOR *******************
 -- La reference de loperation Iterate : $2
 -- Le chemin de loepation Iterate : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

  $2 : $1Iterate
GENERIC MAP ($5, $6, $7)
PORT MAP (
rst     => rst,
clk     => clk,
en      => enable$4,
cpt     => compteur$4,
I_init  => $8,
inport  => $9,
O_si    => $10,
outport => $11
);
')

define(‘_Cnx_diffuse’,
‘--******************* CONNEXION OF Diffuse OPERATOR ********************
 -- La reference de loperation Diffuse : $2
 -- Le chemin de loepation Diffuse : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

  $2 : $1Diffuse
GENERIC MAP ($5, $6)
PORT MAP (
rst    => rst,
inport  => $7,
outport => $8
);
```

```
')

define(‘_Cnx_generic’, ‘ifelse($#,4,‘$3, $4 )’, ‘$3, $4, _Cnx_generic(shift(shift(shift(shift($@)))))’)
')

define(‘_Cnx_port_s_a’, ‘ifelse($#,4,$1 => $2, ‘$1 => $2,
_Cnx_port_s_a(shift(shift(shift(shift($@)))))’)
')

define(‘_Cnx_sens’,
‘--********************* CONNEXION OF SENSOR OPERATOR *******************
 -- La reference de loperation Sensor : $2
 -- Le chemin de loperation Sensor : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

  $2 : $1$6
GENERIC MAP ( _Cnx_generic(shift(shift(shift(shift(shift(shift($@)))))))
PORT MAP (
rst => rst,
clk => clk,
en  => $5,
_Cnx_port_s_a(shift(shift(shift(shift(shift(shift($@)))))))
);
')

define(‘_Cnx_cst’,
‘--******************** CONNEXION OF CONSTANTE OPERATOR ********************
 -- La reference de loperation Constante : $2
 -- Le chemin de loperation Constante : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

  $2 : $1$6
GENERIC MAP ( _Cnx_generic(shift(shift(shift(shift(shift(shift($@)))))))
PORT MAP (
_Cnx_port_s_a(shift(shift(shift(shift(shift(shift($@)))))))
);
')


define(‘_Cnx_actu’,
‘--********************* CONNEXION OF ACTUATOR OPERATOR *******************
 -- La reference de loperation Actuator : $2
 -- Le chemin de loperation Actuator : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

$2 : $1$6
GENERIC MAP ( _Cnx_generic(shift(shift(shift(shift(shift(shift($@)))))))
PORT MAP (
rst => rst,
clk => clk,
en  => $5,
_Cnx_port_s_a(shift(shift(shift(shift(shift(shift($@)))))))
);
')

define(‘_Cnx_port_cout_combi’, ‘ifelse($#,6,‘$2 => $3,
    au$6 => au$6$1,
```

```
    ru$6 => highstate', '$2 => $3,
     au$6 => au$6$1,
ru$6 => highstate,
_Cnx_port_cout_combi($1,shift(shift(shift(shift(shift(shift($@)))))))')
')

define('_Cnx_cout_combi',
'--******* CONNEXION OF CONDITION OUT OPERATOR FOR APPLICATION PUREMENT COMBINATOIRE *********
 -- La reference de loperation Cout : $2
 -- Le chemin de loperation Cout : $3
 -- Cette operation appartient a la frontiere numero $4

  $2_1 : $2
GENERIC MAP ($7,$8,$11,$12,_Cnx_generic_cin_combi(shift(shift(shift(shift(shift(shift
(shift (shift(shift(shift(shift(shift($@))))))))))))
PORT MAP (
rst => rst,
$5  => $6,
$9  => $10,
ad  => highstate,
rd  => rd$2,
_Cnx_port_cout_combi($2,shift(shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift($@))))))))))))
);
')

define('_Cnx_generic_cin_combi', 'ifelse($#,5,'$3, $4 )', '$3, $4, _Cnx_generic_cin_combi(shift(shift(shift(s
')

define('_Cnx_port_cin_combi', 'ifelse($#,6,'$2 => $3,
    rd$6 => rd$6$1,
    ad$6 => highstate', '$2 => $3,
      rd$6 => rd$6$1,
 ad$6 => highstate,
_Cnx_port_cin_combi($1,shift(shift(shift(shift(shift(shift($@)))))))')
')

define('_Cnx_cin_combi',
'--************ CONNEXION OF CONDITION IN OPERATOR FOR APPLICATION PUREMENT COMBINATOIRE ***********
 -- La reference de loperation Cin : $2
 -- Le chemin de loperation Cin : $3
 -- Cette operation appartient a la frontiere numero $4

  $2_1 : $2
GENERIC MAP ($7,$8,$11,$12,_Cnx_generic_cin_combi(shift(shift(shift(shift(shift(shift(shift(shift
 (shift(shift(shift(shift($@))
))))))))))
PORT MAP (
rst => rst,
$5  => $6,
$9  => $10,
ru  => highstate,
au  => au$2,
_Cnx_port_cin_combi($2,shift(shift(shift(shift(shift(shift(shift(shift(shift(
shift(shift(shift($@))))))))))))
);
')
```

```
define(‘_Cnx_generic_cin’, ‘ifelse($#,7,‘$3, $4 )’, ‘$3, $4, _Cnx_generic_cin(shift(shift(shift(shift(shift(s
’)

define(‘_Cnx_port_cin’, ‘ifelse($#,8,‘$2 => $3,
rd$6 => rd$6$1,
ad$6 => $7’, ‘$2 => $3,
rd$6 => rd$6$1,
ad$6 => $7,
_Cnx_port_cin($1,shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))’)
’)

define(‘_Cnx_def_signal’,‘ifelse($#,7,‘ifelse($6,$7,, $6 <= $7;)’,‘ifelse($6,$7,, $6 <= $7;)
_Cnx_def_signal(shift(shift(shift(shift(shift(shift(shift($@)))))))’)
’)

define(‘_Cnx_cin’,

‘--******************  CONNEXION OF CONDITION IN OPERATOR  **************
 -- La reference de loperation Cin : $2
 -- Le chemin de loperation Cin : $3
 -- Cette operation appartient a la frontiere numero $4

  $2_1 : $2
GENERIC MAP ($7,$8,$11,$12,_Cnx_generic_cin(shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift(shift(shift(shift($@))))))))))))))))
PORT MAP (
rst => rst,
$5  => $6,
$9  => $10,
ru  => $13,
au  => au$2,
_Cnx_port_cin($2,shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift($@))))))))))))))))
);
ifelse($13,$14,, $13 <= $14;)
_Cnx_def_signal(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift($@))))))))))))))))

’)

define(‘_Cnx_port_cout’, ‘ifelse($#,8,‘$2 => $3,
au$6 => au$6$1,
ru$6 => $7’, ‘$2 => $3,
au$6 => au$6$1,
ru$6 => $7,
_Cnx_port_cout($1,shift(shift(shift(shift(shift(shift(shift(shift($@)))))))))’)
’)

define(‘_Cnx_cout’,

‘--****************************  CONNEXION OF CONDITION OUT OPERATOR  **************
 -- La reference de loperation Cout : $2
 -- Le chemin de loperation Cout : $3
 -- Cette operation appartient a la frontiere numero $4

  $2_1 : $2
```

```
GENERIC MAP ($7,$8,$11,$12,_Cnx_generic_cin(shift(shift(shift(shift(shift(shift(shift(shift
(shift(shift(shift(shift(shift(shift($@))))))))))))))))
PORT MAP (
rst => rst,
$5  => $6,
$9  => $10,
rd  => rd$2,
ad  => $13,
_Cnx_port_cout($2,shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift($@))))))))))))))))
);
ifelse($13,$14,, $13 <= $14;)
_Cnx_def_signal(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift(shift
(shift($@))))))))))))))))
')

define(`_Cnx_opn',
`ifelse($5,window,
ifelse($4,infini,

`--***************************** CONNEXION OF WINDOW   *********************
 -- La reference de loperation : $2
 -- Le chemin de loperation : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

  $2 : $1$5
GENERIC MAP ( _Cnx_generic(shift(shift(shift(shift(shift($@))))))
PORT MAP (
rst => rst,
en  => enableinf,
clk => clk,
_Cnx_port_s_a(shift(shift(shift(shift(shift($@))))))
);',

`--***************************** CONNEXION OF WINDOW   *********************
 -- La reference de loperation : $2
 -- Le chemin de loperation : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

  $2 : $1$5
GENERIC MAP ( _Cnx_generic(shift(shift(shift(shift(shift($@))))))
PORT MAP (
rst => rst,
clk => clk,
en  => enable$4,
_Cnx_port_s_a(shift(shift(shift(shift(shift($@))))))
);
'),

`--***************************** CONNEXION OF OPERATOR *******************
 -- La reference de loperation : $2
 -- Le chemin de loperation : $3
 -- Cette operation appartient a la frontiere numero $4 donc de uc $4

$2 : $1$5
GENERIC MAP ( _Cnx_generic(shift(shift(shift(shift(shift($@))))))
PORT MAP (
```

```
rst => rst,
_Cnx_port_s_a(shift(shift(shift(shift(shift($@))))))
);')
')

define(‘_Cnx_xpar’,

‘--******************  CONNEXION OF XPAR OR MPAR  *************************
-- La reference de loperation : $2
-- Le chemin de loperation : $3
-- Cette operation appartient a la frontiere numero $4 donc de uc $4
$2 : $5
GENERIC MAP ( _Cnx_generic(shift(shift(shift(shift(shift($@))))))
PORT MAP (
rst => rst,
_Cnx_port_s_a(shift(shift(shift(shift(shift($@))))))
);
')

define(‘_Cnx_mpar’, ‘_Cnx_xpar($@)’
)

define(‘_Cnx_memory’,

‘--***********************  CONNEXION OF MEMORY  ****************************
-- La reference de loperation : $2
-- Le chemin de loperation : $3
-- Cette operation appartient a la frontiere numero $4 donc de uc $4
$2 : $1$6
GENERIC MAP ( _Cnx_generic(shift(shift(shift(shift(shift(shift($@)))))))
PORT MAP (
rst => rst,
en  => $5,
clk => clk,
_Cnx_port_s_a(shift(shift(shift(shift(shift(shift($@)))))))
);
')

define(‘_End’,‘END $1;
-- BEGIN COMPILE SCRIPT
--    vcom definition_$1.vhdl
      undivert(5)
--    vcom $1_toplevel.vhdl
')
```