Réseaux de processus flots de données avec routage pour la modélisation de systèmes embarqués

Anthony Coadou

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS - UFR Sciences
École Doctorale des Sciences et Technologies de l'Information et de la Communication

T H È S E

pour obtenir le titre de

Docteur en Sciences
de l'Université de Nice-Sophia Antipolis

Discipline : Informatique

présentée et soutenue par
Anthony COADOU

RÉSEAUX DE PROCESSUS FLOT DE DONNÉES AVEC ROUTAGE POUR LA MODÉLISATION DE SYSTÈMES EMBARQUÉS

Thèse dirigée par Robert DE SIMONE
soutenue le 3 décembre 2010

Jury :

Pierre BOULET	Professeur, Université de Lille I	Rapporteur
Marc POUZET	Professeur, École Normale Supérieure	Rapporteur
Albert COHEN	Directeur de recherche, INRIA	Examinateur
Nicolas HALBWACHS	Directeur de recherche, CNRS	Examinateur
Robert DE SIMONE	Directeur de recherche, INRIA	Directeur

Résumé

Réseaux de processus flot de données avec routage pour la modélisation de systèmes embarqués

Cette thèse définit un nouveau modèle de calcul et de communication, dénommé *graphe à routage k-périodique* (KRG). Ce modèle, de la famille des réseaux de processus flots de données, admet des aiguillages réguliers des données, explicités par des séquences binaires *k*-périodiques.

Nous étudions les propriétés mathématiques intrinsèques au modèle. Le routage explicite et l'absence de conflit nous permettent d'exprimer algébriquement les dépendances de données, de même que des transformations topologiques préservant le comportement du graphe. Nous montrons ensuite comment ordonnancer le KRG, en associant aux nœuds des horloges *k*-périodiques.

Nous positionnons ensuite notre modèle au sein d'un flot de conception dédié aux applications de traitement intensif de données. Nous montrons en particulier la capacité des KRG à représenter explicitement le parallélisme d'instruction extrait du modèle polyédrique. Nous pouvons alors appliquer un ensemble d'optimisations de bas niveau, sortant du cadre affine du modèle polyédrique. Nous présentons enfin une méthodologie pour l'implantation des KRG, basée sur la conception insensible aux latences.

Mots clés :

Réseau de processus, flot de données, modèle polyédrique, modèle synchrone, conception insensible aux latences, traitement intensif de données.

Équipe-projet INRIA AOSTE[1]
INRIA Sophia Antipolis – Méditerranée
2004 route des Lucioles – BP 93
06902 Sophia Antipolis Cedex
FRANCE

[1] Équipe-projet commune INRIA & I3S (UMR 6070 UNSA/CNRS)

Abstract

Dataflow Process Networks with Routing for Modeling Embedded Systems

This thesis defines a new model of computation and communication, called *k*-periodically routed graph (KRG). This model belongs to the family of dataflow process networks. Data routings are regular and explicit, in the form of *k*-periodic binary sequences.

We study mathematical and behavioral properties of KRGs. Because of explicit routings and conflict-freeness, data dependencies can be expressed in algebraic terms, as well as behavioral-preserving topological transformations. Then we show how KRG schedules can be expressed using *k*-periodic clocks.

We position our model in the heart of a design flow, dedicated to data-intensive processing applications. In particular we show KRG abilities of explicit modeling of instruction-level parallelism, extracted from a polyhedral model. Low level optimizations, out of the scope of polyhedral models, can then be applied to the design. Finally we present a methodology for implementing KRGs, based on latency-insensitive design.

Keywords :

Process network, dataflow, polyhedral model, synchronous model, latency-insensitive design, data-intensive processing.

INRIA project-team AOSTE[2]
INRIA Sophia Antipolis – Méditerranée
2004 route des Lucioles – BP 93
06902 Sophia Antipolis Cedex
FRANCE

[2]Joint project-team INRIA & I3S (UMR 6070 UNSA/CNRS)

Table des matières

Table des figures

Symboles et notations

Dans la suite de ce manuscrit, la plupart des notations utilisées sont introduites dans leur contexte. Nous rappelons ici les notations générales.

Conventions calligraphiques

$\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$...	Ensembles
a, b, c...	Éléments d'ensembles ou scalaires
A, B, C...	Fonctions ou matrices

Opérateurs

$\lvert\mathcal{X}\rvert$	Cardinal de l'ensemble $\mathcal{X}$
$\lvert x\rvert$	Valeur absolue de $x \in \mathbb{R}$
$\lfloor x \rfloor$	Partie entière (fonction plancher) de $x \in \mathbb{R}$
$\lceil x \rceil$	Fonction plafond appliquée à $x \in \mathbb{R}$

Ensembles

$\mathbb{N}$	$\stackrel{\text{def}}{=} \{0, 1, 2, \dots\}$, ensemble des entiers naturels
$\mathbb{N}^*$	$\stackrel{\text{def}}{=} \mathbb{N} \setminus \{0\}$

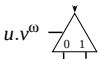
Constantes

ϕ	$\stackrel{\text{def}}{=} \frac{1+\sqrt{5}}{2}$, nombre d'or

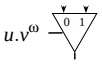
Graphes


Nœud, ou nœud de calcul

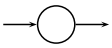
Nœud de copie



Sélection



Fusion


Place

Jeton

Enfin, signalons que nous utilisons le terme *graphe* pour désigner indifféremment des *graphes* et des *hypergraphes*.

Avant-propos

Contexte

Les systèmes embarqués couvrent de nombreux domaines d'applications, et leur utilité s'étend de plus en plus à tous les aspects de la vie quotidienne, des télécommunications (*e.g.* téléphone portable, lecteur multimédia) aux transports (automobile, avionique), en passant par les systèmes de contrôle biométrique et l'imagerie médicale. Tous ces exemples sont des applications effectuant du *traitement intensif de données* : elles utilisent de nombreux filtres numériques, où les opérations sur les données sont répétitives et ont un contrôle limité. Leurs algorithmes peuvent être modélisés aussi bien par des réseaux de blocs de calcul aux états distribués et finis, que par des systèmes de fonctions récursives. En cela, ils répondent aux critères du paradigme de *programmation réactive*.

Les architectures sur lesquelles s'exécutent ces algorithmes sont naturellement parallèles, voire distribuées. Elles sont constituées d'ensembles de composants, comprenant divers processeurs, circuits dédiés et capteurs intelligents. Les modèles de calcul et de communication (*models of computation and communication*, MoCC) associés au domaine de l'embarqué sont donc concurrents et hétérogènes. Pourtant, les langages de programmation les plus largement usités dans le monde de l'informatique, mais aussi et surtout la façon de *penser* et de *concevoir* les algorithmes, sont basés sur le paradigme de programmation impérative et séquentielle. Ce paradigme était initialement destiné à la programmation d'architectures relativement simple et homogène (de type *von Neumann*). Se pose alors un important problème d'unification des paradigmes, ou du moins de rapprochement :

- D'une part, entre des algorithmes de traitement intensif de données, écrits sous la forme de nids de boucles dans des langages de programmation généralistes, et les réseaux de processus flot de données, permettant de modéliser de façon naturelle des systèmes de traitement du signal. Les dépendances de données entre instructions d'un nid de boucles introduisent des contraintes sur l'interconnexion des nœuds de calcul du réseau de processus, et sur le routage des données qui y transitent.
- D'autre part, entre les réseaux de processus flot de données et de la conception réactive synchrone et polychrone, dans laquelle les horloges d'activation des différents blocs de calcul sont rendues explicites, par le biais d'ordonnancements statiques et réguliers. Un réseau de processus est alors transposé dans le modèle synchrone, *n*-synchrone ou polychrone, en vue de son ordonnancement.

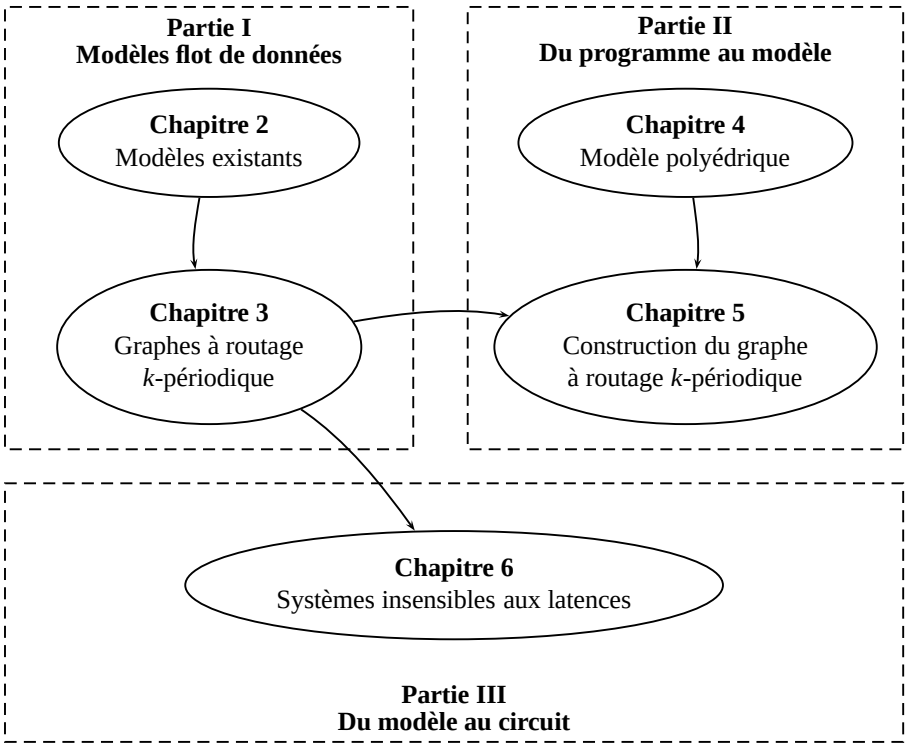


FIG. 1: Dépendances des différents chapitres.

Notre thèse *in a nutshell*

Dans ce cadre, nous introduisons les graphes à routage k-périodique (KRG), modèle central de cette thèse. Ce modèle flot de données allie la simplicité du routage des données *à la BDF*[3] avec les aspects explicites des séquences de routage de CSDF[4]. De plus, le routage des données y est représenté sous la forme de séquences binaires ultimement périodiques, utilisées dans le modèle n-synchrone à des fins d'ordonnancement ; ces séquences binaires nous permettent d'exprimer algébriquement des transformations topologiques, tout en garantissant la préservation du comportement du graphe.

Nous étudions dans un premier temps les propriétés théoriques fondamentales des KRG. Puis nous considérons à la suite leur capacité à encoder les résultats de la parallélisation de nids de boucles, et leur capacité à soutenir l'étude d'ordonnancement vers des modèles réactifs synchrones et polychrones. Les contributions principales de ces différents chapitres sont détaillées ci-dessous. L'ordre de lecture des différents chapitres est quant à lui illustré en FIG. 1.

Graphes à routage k-périodique

Dans une première partie, le chapitre 3 traite des graphes à routage k-périodique, de leurs propriétés et de leurs transformations, ainsi que de leurs relations par abstraction ou expansion avec d'autres modèles. Les principaux résultats sont :

– une abstraction d'un KRG vers le modèle SDF[5], en intégrant les comportements sur leurs pé-

[3]Pour *boolean-controlled dataflow*, *cf.* sous-section 2.4.1.
[4]Pour *cyclo-static dataflow*, *cf.* sous-section 2.3.1.
[5]Pour *synchronous data flow*, *cf.* sous-section 2.2.4.

riodes et en préservant les débits des nœuds (*cf.* sous-section 3.1.4, page 62) ;
– un dépliage vers le modèle des (quasi)-graphes marqués, nous permettant de fixer une sémantique de référence aux KRG (*cf.* sous-section 3.2.6, page 73) ;
– une expansion de l'interconnexion d'un KRG, départageant les canaux de communication. Cette expansion permet d'établir la complétude de l'axiomatique des transformations locales de la topologie (*cf.* sous-section 3.3.4, page 86).

Liens avec le modèle polyédrique

Dans une seconde partie, nous considérons l'usage des KRG pour exploiter les résultats de la parallélisation d'un nid de boucles. De nombreux travaux proposent de représenter un programme de traitement intensif de données sous la forme de graphe de dépendance polyédrique, pour en donner une représentation géométrique. Cette forme s'avère particulièrement utile pour y appliquer un ensemble de transformations visant à en extraire et manipuler du parallélisme d'instruction. Cependant, seules des transformations linéaires s'appliquent dans le cadre du modèle polyédrique, ce qui constitue une contrainte forte sur l'exploitation de ce parallélisme. D'autre part, le problème de sa représentation sous forme opérationnelle se pose. Une solution généralement adoptée par la communauté de la compilation consiste à générer, à partir du modèle polyédrique, du code impératif annoté de directives de parallélisation.

Nous montrons au chapitre 5 les capacités des KRG à modéliser ces résultats. Ils offrent une représentation fidèle du parallélisme extrait par ces méthodes existantes. Nous discutons brièvement la possibilité d'enchaîner à leur suite d'autres transformations, décrite précédemment au chapitre 3. Ces transformations, sortant du cadre linéaire, nous permettent d'affiner les résultats de la parallélisation et de la transformation de l'application.

Architecture d'un système-sur-puce avec routage

Dans une troisième partie, nous abordons le problème de l'implantation de systèmes modélisés par des KRG. La méthodologie insensible aux latences fut développée en vue de garantir la correction fonctionnelle d'un système dont les latences de calcul et de communication ne permettraient pas de respecter les hypothèses synchrones. Une première approche consiste à resynchroniser les signaux à la volée, grâce au protocole dit *insensible aux latences*. Une autre solution consiste à calculer des ordonnancements statiques du système, tout en prenant en compte ces latences.

Au chapitre 6, nous étudions différentes implantations de l'état de l'art, destinées exclusivement à des flots de données purs, dépourvus de routage variable. Nous en proposons des extensions visant à implanter des KRG.

Chapitre 1

Introduction

> Les sciences n'essaient pas d'expliquer, elles essaient à peine d'interpréter, elles font principalement des modèles. Par modèle on entend une construction mathématique qui, par l'addition de quelques interprétations verbales, décrit des phénomènes observés. La justification d'une telle construction mathématique est uniquement et précisément qu'elle est censée marcher.
>
> John von Neumann, Method in the Physical Sciences. *The Unity of Knowledge*, L. Leary (éd.), 1955.

Dans sa leçon inaugurale au Collège de France, intitulée *Pourquoi et comment le monde devient numérique* [25], Berry explique que l'on a trop souvent associé *informatique* avec *ordinateur*. Pourtant, le monde dans lequel nous vivons s'est considérablement enrichi d'une multitude de dispositifs de traitement de l'information : les télécommunications, le multimédia ou les transports n'en sont que des exemples parmi tant d'autres. On parle alors d'informatique ubiquitaire, et même d'*everyware* [104]. Les systèmes embarqués y occupent une place prépondérante, dans le sens où ils ont envahi tous les aspects de la vie quotidienne. Ils interagissent continuellement avec leur environnement par le biais de divers capteurs intelligents, filtres numériques et contrôleurs. Et deviennent d'autant plus complexes.

Aussi le nombre de transistors par circuit intégré a jusqu'à présent suivi la loi de Moore, doublant en moyenne tous les deux ans. Cette croissance exponentielle de la complexité des systèmes demande à automatiser le plus possible leur conception. Ainsi, à titre de comparaison, le premier microprocesseur commercialisé en 1971, l'Intel 4004, comptait 2 300 transistors sur à peine 12 mm^2 [155] ; quarante ans plus tard, un Intel Xeon 7500 en compte un million de fois plus, pour une densité multipliée par près de 15 000 [119]. Si une poignée d'ingénieurs a pu dessiner le premier à la main, et au niveau porte logique (*gate level*), cela fait longtemps qu'il est devenu indispensable de rehausser le niveau d'abstraction.

Dans l'avant-propos de $A = B$ [134], Knuth écrit : «*la science est tout ce que nous comprenons suffisamment bien pour l'expliquer à un ordinateur. L'art, c'est tout le reste*». Aussi, par *niveau d'abstraction*, on entend *modèle* : une abstraction mathématique de la réalité, trop complexe à appréhender dans son ensemble, nous permettant de nous concentrer sur un, voire plusieurs concepts. Dans le contexte des systèmes sur puces, et d'autant plus pour les systèmes embarqués, les flots de conception font appel à des modèles dont la sémantique est suffisamment riche pour représenter des comportements complexes, et en même temps suffisamment contrainte (*i.e.* absence de conflit) pour en dégager formellement des propriétés : *e.g.* absence de blocage, dimensionnement des mémoires, *etc*.

Depuis quelques années, les outils de synthèse de haut niveau tendent à se faire adopter[1] par l'industrie pour la conception assistée par ordinateur d'électronique (*electronic design automation* – EDA). Plutôt que de concevoir le système au niveau transfert de registres (*register-transfert level* – RTL), le développeur en donne une description fonctionnelle ; cette approche offre l'énorme avantage de décrire le système sous une forme unique, indépendamment de l'architecture cible. Dans un second temps, le code est analysé, contraint par les ressources matérielles et l'ordonnancement ; les parties de contrôle de l'application sont généralement compilées pour être exécutées par un processeur, tandis que les parties de traitement intensif de données sont plutôt destinées à des accélérateurs dédiés.

En effet, Girbal *et al.* ont montré que les portions de code effectuant des traitements répétitifs avec un contrôle limité, dans des applications scientifiques ou de traitement du signal, représentent une importante part du temps de calcul sur CPU ou DSP [99]. On retrouve cette propriété comme un corollaire du principe de Pareto : 20% des instructions d'un programme coûtent 80% de son temps d'exécution. Ainsi, la synthèse de circuits dédiés a encore de beaux jours devant elle : si un FPGA peut se montrer 20 fois plus rapide qu'un DSP pour certaines applications de traitement du signal [21], une implantation ASIC est en moyenne 40 fois plus petite, 3 fois plus rapide, et consomme 12 fois moins de puissance dynamique que son équivalent FPGA, à technologie de gravure équivalente [140].

Dans ce manuscrit, nous nous intéressons en particulier à la modélisation et à la conception de telles applications de traitement intensif de données. Dans la section suivante, nous faisons un tour d'horizon des différentes approches utilisées dans les mondes industriels et académiques pour la synthèse de haut niveau. Nous définissons et sélectionnons ensuite les éléments d'un flot de conception spécifique. Nous décrivons alors comment nos modèles s'y intègrent.

1.1 Modèles et langages pour la synthèse de haut niveau

1.1.1 Langages généralistes

Parmi les solutions actuellement disponibles sur le marché, les outils générant du code RTL à partir d'une description dans un langage généraliste sont certainement les plus plébiscités. Le langage de support est souvent le C-ANSI, voire le C++, et leurs dérivés *ad hoc* (SystemC, BrookGPU, Cg, CUDA, *etc*). La conception du matériel s'apparente alors à du développement logiciel dans un langage largement utilisé. Les principaux acteurs de l'industrie de l'EDA ont suivi cette approche : Catapult C (Mentor Graphics), Cynthesizer (Forte Design Systems), Synphony C Compiler (Synopsys), PowerOpt (ChipVision Design Systems), C-to-Silicon (Cadence), ou encore Impulse C (Impulse Accelerated Technologies), pour ne citer qu'eux.

Toutefois, des langages aussi généralistes que C ou C++ posent plusieurs problèmes pour la description de matériel. À commencer par le fait que, jusqu'à présent, aucune sémantique formelle ne fait consensus : la sémantique du programme repose donc en partie sur l'interprétation qu'en fait le compilateur.

De plus, l'expressivité de C est beaucoup trop large par rapport aux langages traditionnels de description de matériel, tels que Verilog ou VHDL. La sémantique de C doit être bridée, de façon à y interdire, par exemple, les allocations dynamiques de mémoires et les manipulations de pointeurs, qui sont des concepts logiciels sans pendant matériel. On parle alors de C *synthétisable*, dont les nids de boucles sont un exemple typique.

Enfin, les langages généralistes les plus courants sont impératifs et séquentiels ; il est alors nécessaire de conduire une analyse de code assez poussée pour en extraire le parallélisme d'instructions et

[1] Bien que leur principe ne soit pas récent, il est l'objet de recherches académiques depuis le début des années 1980 [114].

de données.

1.1.2 Systèmes d'équations récurrentes et modèle polyèdrique

Les systèmes d'équations récurrentes [128], et par extension le modèle polyédrique [149], offrent une représentation géométrique de l'application. Elle généralise la représentation des domaines des variables, souvent représentés par des tableaux, à des polyèdres quelconques dans l'espace. Une case du tableau y est alors assimilée à un point à coordonnées entières à l'intérieur du polyèdre. Les dépendances entre données y sont également représentées sous forme de polyèdres.

Cette approche permet d'utiliser la puissance de résultats et d'outils de géométrie algébrique. Cependant, les problèmes à résoudre étant pour la plupart *difficiles*, le modèle polyédrique se limite au cas linéaire : les bornes des polyèdres, les dépendances et les transformations sont fonctions affines des paramètres. On se ramène ainsi à des problèmes d'algèbre linéaire, pouvant être résolus de façon relativement efficace.

En particulier, le langage Alpha et son environnement MMAlpha ont été conçus pour la manipulation d'applications exprimées dans le modèle polyédrique [79, 163, 233].

1.1.3 Réseaux de processus

Les réseaux de processus, quant à eux, modélisent le système sous forme d'un graphe. Les sommets sont des tâches, de différents niveaux de granularité selon les modèles, et les arcs sont des liaisons point-à-point, éventuellement munies de mémoires.

Cette approche est certainement l'une des plus naturelles et intuitives pour la description de flots de données. Elle bénéficie en outre d'une riche théorie, issue des résultats de la théorie des graphes, des algèbres de processus ou encore de l'ordonnancement. Différents outils se basent sur ces modèles : K-Passa [35], Ptolemy [44], ou encore StreamIt [102], pour n'en citer que quelques-uns issus du monde académique.

Array-OL est un langage de description d'applications flots de données multidimensionnelles [41]. Il présente deux niveaux de modélisation : (i) l'un global, en représentant l'application sous la forme d'un réseau de processus, consommant et produisant des tableaux multidimensionnels ; (ii) l'autre local, exprimant les relations entre les cases des tableaux par des systèmes d'équations. En cela, Array-OL se situe à cheval entre les réseaux de processus et les systèmes d'équations récurrentes. Nous le rangeons plutôt dans la première famille pour deux raisons : (i) les tableaux et dépendances de données manipulées par Array-OL sont différents de ceux du modèle polyédrique, bien qu'ils soient tout deux relativement proches [188], et (ii) les auteurs et contributeurs d'Array-OL ont plutôt tendance à comparer leur approche avec celle de GMDSDF[2] [78, 100].

1.1.4 Langages synchrones et polychrones

Bien que les modèles synchrones et polychrones furent initialement conçus pour la programmation et la synthèse de systèmes de contrôle-commande, ils rejoignent sous de nombreux aspects le principe des systèmes d'équations récurrentes et méritent, à ce titre, d'être considérés pour la description d'application de traitement intensif [23]. Quoiqu'à la différence des systèmes d'équations, ces langages ne représentent généralement que des variables à une seule dimension (le temps), ou au plus deux si l'on considère la concurrence (espace et temps).

[2] Pour *generalized multidimensional synchronous dataflow* [176].

Des langages tels que Lustre, Lucid Synchrone ou Signal offrent à la fois une sémantique formelle et une expression naturelle des traitements flots de données [179, 192]. La notion d'horloge unique du paradigme synchrone est particulièrement adaptée pour représenter des applications de bas niveau, proches du matériel, tandis que les modèles polychrones permet de remonter la description à un niveau plus comportemental. Aussi, à la façon des réseaux de processus, les modèles synchrones et polychrones offrent une double représentation, textuelle et graphique : Scade pour Lustre, SyncCharts pour Esterel, Polychrony pour Signal.

1.1.5 Approches mixtes

Un certain nombre de travaux se sont intéressés à combiner plusieurs des approches précédemment énoncées. L'intérêt de ces approches mixtes consiste à palier les inconvénients d'un modèle par les avantages d'un autre. Ces combinaisons peuvent être aussi bien verticales (*e.g.* abstraction contre raffinement) que horizontales (*e.g.* flot de contrôle contre flot de données).

Nids de boucles + polyèdres

La combinaison de modèles la plus courante en matière de compilation consiste à utiliser le modèle polyédrique pour l'analyse et la parallélisation de nids de boucles. Le développeur écrit l'application sous forme d'un nid de boucles en langage C, puis le code source est analysé et abstrait en un modèle polyédrique. C'est alors au sein du modèle polyédrique que sont effectuées les transformations de l'application.

Cette approche utilise conjointement la simplicité d'un langage tel que C, largement répandu dans l'industrie, avec l'expressivité du modèle polyédrique et la puissance des outils d'algèbre linéaire. En contrepartie, elle souffre des inconvénients des deux approches : la sémantique de C n'est pas formelle, et le modèle polyédrique ne supporte que des domaines, dépendances et transformations linéaires.

Cette approche est l'objet de recherches depuis les années 1980, qui ont donné naissance à plusieurs compilateurs académiques, tel que LeTSeE [194], LooPo [108], Pluto [32] ou WrapIt [65], aussi bien que commerciaux : R-Stream (Reservoir Labs), AutoPilot (AutoESL), Synphony C Compiler (ex-PICO, Synopsys) [2, 207, 208].

Polychrone + polyèdres

L'utilisation conjointe de MMAlpha et Polychrony est un exemple typique de combinaison des modèles polychrones et polyédriques [216]. Le flot de données de l'application, dont le contrôle est réduit au minimum, est décrit sous forme polyédrique dans le langage Alpha. La logique de contrôle, quand à elle, est programmée en Signal.

Cette approche est un exemple de combinaison horizontale des deux modèles, où flots de données et flots de contrôle sont décrits de façon distincte, chacun dans un langage approprié. L'énorme avantage de cette approche est précisément de séparer dès la conception les données du contrôle : d'une part elle facilite l'analyse du code, et d'autre part elle permet d'utiliser un paradigme de programmation approprié pour chaque partie. En contrepartie, elle demande un effort supplémentaire au développeur qui utilise deux outils au lieu d'un, ce qui peut sembler fastidieux pour un non-spécialiste, du moins au premier abord.

Nids de boucles + polyèdres + réseaux de processus

Des travaux sont menés depuis près de quinze ans à l'Université de Leyde autour du projet Compaan et de son successeur Pn [118, 133, 181, 218, 219, 226, 230]. Ces outils prennent en entrée un code source Matlab ; celui-ci est abstrait en un modèle polyédrique, qui est ordonnancé et transformé. Une fois que ces transformations ont convergé, un réseau de processus est généré à partir du modèle polyédrique, sur lequel est appliquée une nouvelle série d'optimisations.

L'originalité de ces travaux est de donner une vue globale d'un flot de conception à différents niveaux d'abstractions. Contrairement aux précédentes approches, les transformations sur les modèles sont effectuées en deux temps : d'abord des transformations de haut niveau sur le modèle polyédrique, puis des transformations de bas niveau sur le modèle SBF[3]. Il est intéressant de constater l'adéquation entre, d'une part, les transformations à effectuer sur l'application et, d'autre part, les modèles supports à chaque étape.

Notons que les travaux de Derrien [75] et Thies *et al.* [224] vont dans ce sens, en établissant des liens entre le modèle polyédrique et les réseaux de processus.

1.1.6 Observations

Chaque langage ou modèle précédemment cité comporte un certain nombre d'avantages et d'inconvénients. Nous pouvons en déduire des principes très généraux :

C'est à l'outil de s'adapter au développeur, et non l'inverse. Un algorithme optimal pour un processeur séquentiel n'a *a priori* rien a voir avec l'algorithme naturellement parallèle implanté sous forme de circuit électronique. On peut donc légitimement se poser la question de la pertinence de C ou C++, par exemple, pour décrire des applications matérielles. Pourtant, ces langages se sont indéniablement imposés dans l'industrie pour la synthèse de haut niveau. La raison est simple : ils sont maîtrisés par un grand nombre de développeurs, et disposent d'environnements de développement évolués et de nombreuses bibliothèques.

Le modèle polyédrique est approprié pour modéliser un flot de données à contrôle affine. À partir d'un code source écrit dans un langage généraliste tel que C ou Fortran, les compilateurs traditionnels construisent des représentations syntaxiques dont la granularité de l'ordre de l'*instruction*. Elles supportent des transformations simples, comme la propagation de constantes ou la recherche de sous-expressions communes. Pourtant, des optimisations plus évoluées, portant notamment sur les boucles, y sont fastidieuses. Bastoul note que «*les transformations les plus intéressantes demandent à modifier l'ordre d'exécution du programme, et cela n'a rien à voir avec la syntaxe*» [18]. Dans ce contexte, l'intérêt du modèle polyédrique pour représenter et analyser des parties à contrôle statique de flots de données n'est plus à démontrer, et a fait l'objet de nombreux travaux précédemment cités.

Le modèle synchrone est le standard pour la conception de circuits électroniques numériques. La très grande majorité des circuits numériques actuels sont dirigés par des horloges. À chaque cycle d'horloge, le circuit échantillonne l'ensemble de ses entrées, et émet des signaux sur l'ensemble de ses sorties. C'est donc, par définition, un système synchrone. Par *émettre*, il faut comprendre qu'un signal a, à chaque instant, un et un seul *état*. Un signal *pur* peut être soit absent (tension nulle en sortie du circuit), soit présent (tension non-nulle).

Les approches mixtes permettent de résoudre beaucoup de problèmes. Chaque modèle a une expressivité et une granularité propre. Bien que tous ces modèles aient globalement la même vocation

[3] Pour *stream based functions*, *cf.* sous-section 2.4.2.

in fine, à savoir la description d'une application flot de données, ils la représentent sous des angles différents. De ce fait, certaines transformations s'expriment plus naturellement dans un modèle que dans un autre. Par analogie avec le traitement du signal, si certaines transformations sont faciles à appliquer dans le domaine temporel, d'autres le sont beaucoup plus dans le domaine fréquentiel. Ainsi, procéder par raffinements successifs et appliquer, à chaque étape, les transformations adéquates permet de simplifier, et parfois optimiser, le flot de conception.

1.2 Notre approche

À partir des points précédemment énoncés, nous pouvons imaginer un flot de conception. C'est dans ce cadre que nous intégrons notre propre modèle, visant à apporter un nouvel éclairage à certains travaux existants.

La méthodologie présentée par la suite consiste en un pipeline de transformations entre modèles, chacun adapté à une étape de la conception. L'objectif est de tirer profit des avantages de chaque modèle, tout en minimisant l'impact de leurs inconvénients sur le flot de conception : chaque transformation est alors effectuée au niveau où elle s'exprime de la façon la plus naturelle. Nous affirmons que cette démarche est pertinante pour trois raisons : (i) une composition judicieuse de modèles existants n'est pas un choix évident, (ii) notre choix est motivé par la connaissance des points forts et faibles des différentes approches existantes, et enfin (iii) elle met en relation les travaux issus de différentes communautés scientifiques pour les adapter à un problème commun. De plus, les approches mixtes précédemment citées ont fourni des résultats performants et prometteurs ; notre contribution se positionne donc en appui de ces travaux.

1.2.1 Adéquation des modèles

Nous envisageons une approche où les modèles sont intégrés verticalement, par abstractions ou raffinements successifs. Comme nous en avons précédemment discuté, nous partons du principe que le développeur écrit son application dans un langage généraliste, sous la forme d'un ou plusieurs nids de boucles. Par extension, nous pouvons envisager que ces nids de boucles appartiennent à différents processus d'un réseau *à la Kahn*, communiquant via des canaux point-à-point. Ce formalisme d'entrée nous permet de donner une description de haut niveau de l'application, tout en réduisant la complexité des calculs ultérieurs, comme l'a montré Feautrier [88].

Selon l'état de l'art, le modèle polyédrique est l'un des plus appropriés pour analyser et transformer un flot de données multidimensionnelles par des outils d'algèbre linéaire ; les domaines des variables et les dépendances sont définis selon des fonctions affines. De plus, il permet de représenter un flot de données *paramétré*, contrairement à la plupart des modèles déterministes : les bornes et relations entre données dépendent de paramètres inconnus à la compilation. Deux questions se posent alors :

1. Tout d'abord, le modèle polyédrique offre une représentation fonctionnelle des données et de leurs dépendances ; il n'a cependant pas de sémantique opérationnelle. Vers quel modèle pouvons nous exporter l'application de façon à représenter explicitement et exploiter le parallélisme extrait du modèle polyédrique ?
2. Ensuite, le modèle polyédrique se limite au cas affine. Est-il possible d'effectuer (i) des transformations à gros grain, linéaires, dans le modèle polyédrique, puis (ii) en raffiner les résultats par des transformations à grain fin, plus complexes ?

Afin de répondre au mieux à ces questions, nous introduisons les graphes à routage *k*-périodique (KRG). Ce sont des réseaux de processus flot de données, sans conflit et déterministe, alliant la simplicité du routage de données de BDF [45] avec le principe du routage explicitement guidé par des séquences de routages, comme en CSDF [31, 81]. Les opérations sont effectuées par des nœuds de calcul, consommant et produisant des données abstraites sous la forme de jetons. Routages et ordonnancements sont exprimées par des séquences binaires ultimement périodiques, issues de la théorie du *n*-synchrone [62, 63].

L'intérêt des routages et ordonnancements ultimement périodiques est double. D'une part, il nous permet d'exprimer algébriquement un ensemble de propriétés et transformations topologiques, se ramenant à du simple calcul binaire. Ces transformations s'appliquent à un seul ou un sous-ensemble de sommets du graphe, et non à la totalité ; en ce sens, elles sont plus fines que celles du modèle polyédrique, car non-linéaires. D'autre part, ils nous permettent de revisiter les ordonnancements linéaires du modèle polyédrique. Enfin, un ordonnancement *au plus tôt* leur confère une sémantique synchrone, proche de l'architecture ; la transcription dans un langage synchrone, tel que Lustre ou Esterel, est alors possible.

1.2.2 Intégration dans un flot de conception

Le flot de conception, suivant lequel l'application est transformée d'un modèle à l'autre, est représenté en FIG. 1.1. Dans l'idée, il s'inspire de celui des projets Compaan et Pn : à partir d'un nid de boucles, nous construisons un modèle polyédrique, puis un réseau de processus. Si les premières moitiés des deux flots sont identiques, les deux différences majeures se situent au niveau du réseau de processus et du circuit généré. Notre flot se décompose comme suit :

Décomposition analytique, analyse des bornes et dépendances

Un nid de boucles ne peut pas être transposé tel quel dans le modèle polyédrique, il doit d'abord subir plusieurs transformations intermédiaires. Il est tout d'abord transformé en un arbre de syntaxe abstraite, puis sous forme à assignation unique. De cette façon, les dépendances de données induites par les réécritures dans les mêmes variables sont éliminées. Ensuite, une analyse des domaines de définition des variables et de leurs dépendances est nécessaire, afin de construire les polyèdres correspondants.

Différents outils permettent de construire un modèle polyédrique à partir d'un nid de boucles : *e.g.* MatParser (utilisé par Compaan) [132, 133], Clan [19] ou LooPo [108].

Extraction du parallélisme, transformations linéaires

L'extraction du parallélisme s'effectue au sein du modèle polyédrique. Pour cela, nous recherchons les transformations linéaires à appliquer au modèle : il nous faut distinguer les opérations devant être effectuées en séquence, à cause d'une relation de dépendance, de celles pouvant être effectuées simultanément.

La première étape consiste à borner l'ensemble des transformations valides, à savoir celles qui respectent les contraintes imposées par les domaines de définitions et les dépendances de données.

Ensuite, nous y recherchons la *meilleure* transformation possible, selon une métrique. Cette métrique est exprimée sous la forme d'une fonction linéaire, à un certain critère (*e.g.* tailles des mémoires, débit du système, *etc*). Le choix de la *meilleure* transformation revient donc à résoudre un problème de recherche opérationnelle, où l'on calcule la transformation permettant de maximiser ou minimiser

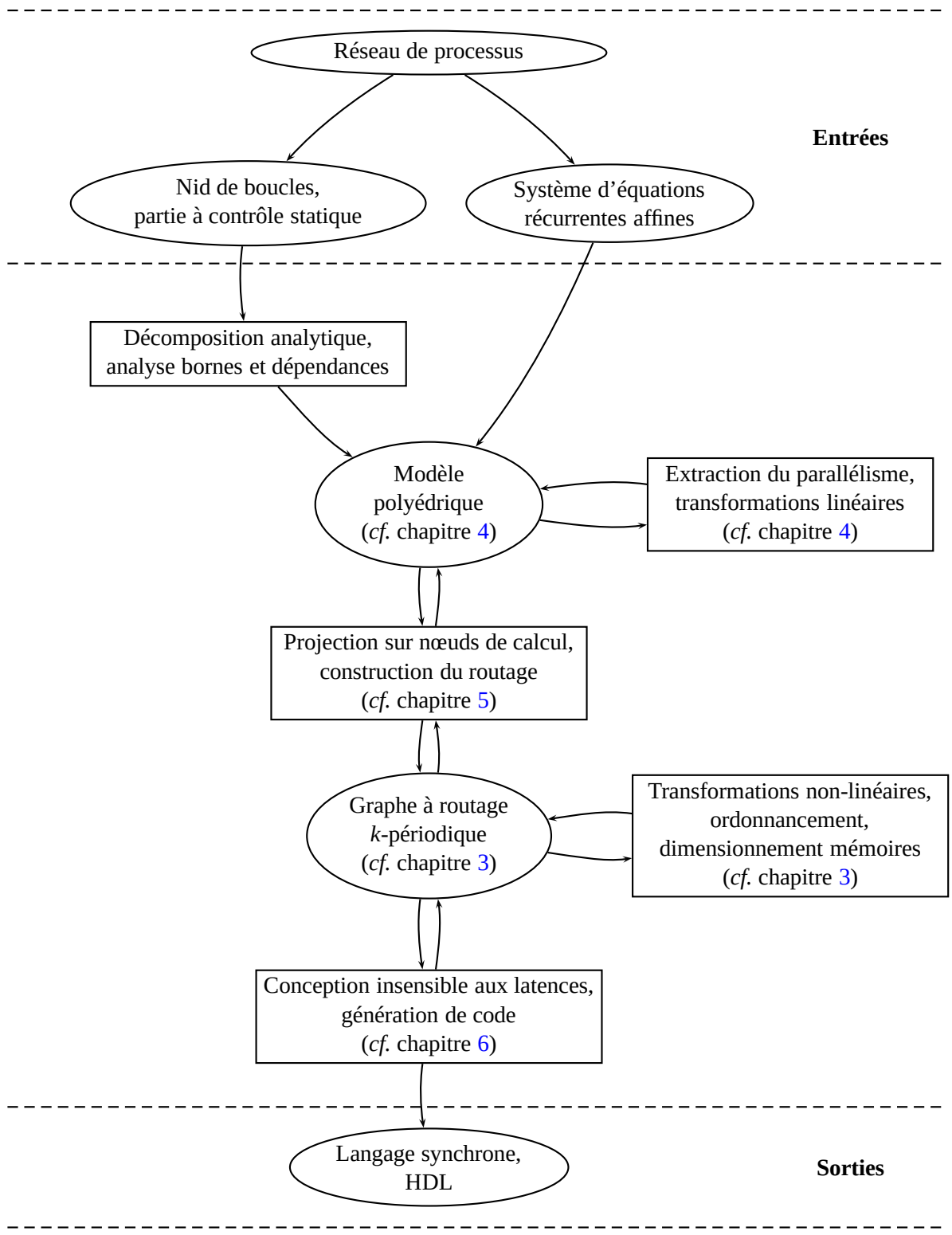


FIG. 1.1: Schéma fonctionnel du flot de conception.

la métrique. Le problème est alors résolu par un solveur de programme linéaire en nombres entiers. Cette technique de parallélisation et d'optimisation a montré son efficacité au travers des nombreux travaux dont elle fut l'objet [32, 87, 105, 109, 152, 153, 182, 225].

Projection sur nœuds de calcul, construction du routage

Le parallélisme extrait du modèle polyédrique infère un certain nombre de sous-ensembles d'opérations pouvant être exécutées simultanément. Grâce à cela, nous projetons ces opérations sur un graphe à routage *k*-périodique : chaque opération y est associée à la production d'un jeton par un nœud de calcul. Ainsi le parallélisme d'instruction et de données y est explicitement représenté.

L'interconnexion entre nœuds de calcul est déterminée selon les dépendances de données : si deux opérations dépendent l'une de l'autre, alors il existe un chemin correspondant dans le KRG, permettant d'acheminer un jeton d'un nœud de calcul à l'autre.

Une première différence entre notre approche et Compaan réside dans le niveau d'abstration conféré aux deux réseaux de processus (KRG dans notre cas, SBF pour Compaan) lors de leur construction à partir du modèle polyédrique.

Par exemple, pour calculer les valeurs de quatre tableaux multidimensionnels, Compaan construit un graphe SBF à quatre nœuds de calcul : chaque nœud de calcul correspond à un nid de boucles séquentielles permettant le parcours d'un tableau. Et ce, même si plusieurs cases du tableau peuvent être calculées simultanément ; cela revient à perdre une partie du parallélisme extrait du modèle polyédrique. Dans le flot de conception de Compaan, cela correspond à l'étape de *linéarisation* [133, 226]. Autrement dit, le graphe SBF ainsi généré n'exploite que du parallélisme de données : deux instructions indépendantes peuvent être évaluées en parallèle.

Dans notre cas, nous pouvons construire un KRG contenant autant de nœuds de calcul que le nombre maximum de cases de tableau pouvant être calculées simultanément. C'est-à-dire qu'en plus du parallélisme de données, nous exploitons également le parallélisme d'instruction : si deux occurrences d'une même instruction sont indépendantes, alors elles peuvent être évaluées en parallèle.

Transformations non-linéaires, ordonnancement, dimensionnement des mémoires

Les propriétés algébriques des KRG permettent d'affiner les résultats des transformations linéaires du modèle polyédrique. Tout d'abord, nous pouvons appliquer des transformations topologiques visant à «équilibrer» les flots de jetons au sein du KRG, tout en préservant sa fonctionnalité. Par exemple, certains nœuds de calculs peuvent être plus ou moins utilisés à l'exécution ; il est alors utile, selon le cas de figure, de (i) rerouter des jetons d'un nœud de calcul vers un autre pour équilibrer leur charge, (ii) fusionner plusieurs nœuds de calcul peu utilisés en un seul, et entrelacer les flots, ou au contraire (iii) dupliquer un nœud de calcul et éclater les flots de façon à alléger la charge de chacun. Ces différentes optimisations permettent d'optimiser localement le réseau de processus.

Le fait d'associer aux nœuds de calcul une règle de production et consommation *au plus tôt* confère au graphe une sémantique *n*-synchrone. Du fait que le routage des jetons est sans conflit et ultimement périodique, et que nous considérons des latences constantes, le graphe peut être ordonnancé statiquement. Les capacités des mémoires peuvent alors être dimensionnées de façon optimale.

Conception insensible aux latences, génération de code

Enfin, le KRG synchrone peut facilement s'exporter vers un programme synchrone, et en particulier *n*-synchrone. Les nœuds de calculs du KRG se comportent alors comme des modules synchrones. La réécriture en Lucy-*n* [158], par exemple, est immédiate.

Les nœuds de calculs peuvent se trouver désynchronisés lorsque les latences de communications ne permettent pas d'acheminer les données de l'un à l'autre en l'espace d'un cycle d'horloge. Pour cette raison nous appliquons au KRG une méthodologie dite de *conception insensible aux latences*. Cette méthodologie garantit la préservation de la fonctionnalité de l'application. Son comportement est alors globalement asynchrone, et localement synchrone.

L'implantation du LID peut se faire soit par le biais d'un protocole dynamique, soit par ordonnancement statique. Dans le premier cas, le protocole coupe l'horloge des modules qui ne disposent pas de données à l'ensemble de leurs entrées, et rétropropage un signal d'arrêt en cas de congestion. Dans le second cas, les ordonnancements des nœuds sont calculés statiquement à la compilation. Un contrôleur, faisant office d'oracle, active et désactive les nœuds lorsque nécessaire.

1.2.3 Contributions et plan

Première partie : modèles flot de données

Le chapitre 2 établit essentiellement des rappels sur les principaux modèles flots de données de l'état de l'art. Nous rappelons tout d'abord des propriétés fondamentales sur les réseaux de processus, concernant la propriété d'absence de conflit, et toutes ses conséquences bénéfiques : (i) (*confluence* et *équité*) une tâche effectuable sera forcément effectuée et ne peut être repoussée indéfiniment ; (ii) (*unicité du comportement*) toutes les exécutions possibles du modèle ne sont que les ordonnancements différents d'un même comportement partiellement ordonné. Puis nous retraçons les principaux résultats sur les graphes marqués, sur lesquels nous nous appuierons par la suite. Nous proposons un algorithme pour calculer les dimensions minimales des places tout en maximisant le débit du graphe. Ensuite nous présentons les deux grandes familles de modèles qui étendent les graphes marqués par des routages variables, statiques puis dynamiques. Enfin, nous évoquons des modèles plus complexes, faisant intervenir routage paramétré et hiérarchie.

Le chapitre 3 présente le modèle central de cette thèse, les graphes à routage k-périodique (KRG). Initialement introduits par Boucaron [33], nous en formalisons leur définition et leur sémantique dans une première section. Nous montrons comment vérifier leur caractère borné, au moyen d'une abstraction en graphe SDF. Nous abordons aussi la question de la vivacité du graphe. Dans une seconde section, nous étudions les dépendances de données au sein du KRG. Nous construisons une première représentation des dépendances de données au moyen d'un *graphe de dépendance étendu*, potentiellement infini. Nous définissons alors deux relations d'équivalences entre données, selon lesquelles nous pouvons borner notre étude à un domaine fini, représenté par un *graphe de dépendance réduit*. Cette dernière représentation des dépendances nous permet alors de construire, pour tout KRG, sa forme normale : nous fixons ainsi une sémantique de référence aux KRG, en établissant leur lien avec les graphes marqués. Les deux sections suivantes sont consacrées aux transformations de la topologie du KRG, tout en préservant leur fonctionnalité. Nous construisons une forme expansée de l'interconnexion du graphe, et montrons la complétude des transformations sur le routage. La dernière section présente l'adjonction d'une sémantique synchrone aux KRG. Nous pouvons alors discuter de leur ordonnancement et du dimensionnement des mémoires.

Seconde partie : du programme au modèle

Le chapitre 4 rappelle les principes du modèle polyédrique, ainsi que son utilité pour la compilation d'applications flot de données. La première section présente les deux approches usuelles à partir desquelles est construit le modèle polyédrique, à savoir les systèmes d'équations récurrentes et les nids de boucles. Ensuite, nous expliquons les différents concepts du modèle polyédrique, représentés sous

la forme d'un *graphe de dépendance réduit polyédrique* (GDRP). Enfin nous rappelons les différents résultats de l'état de l'art sur la parallélisation de flots de données au sein du modèle polyédrique.

Le chapitre 5 détaille la construction d'un KRG à partir d'un GDRP. La première section explique comment déterminer le nombre de nœuds de calcul nécessaires à l'exécution parallèle des opérations, et leur projection sur ces nœuds de calcul. La seconde section expose la construction de l'interconnexion des nœuds de calcul. Nous calculons le routage nécessaire à la scission et à la fusion de flots de jetons, ainsi qu'à leur permutation pour une consommation dans le désordre. Enfin nous discutons, dans une dernière section, des avantages et inconvénients de notre solution, ainsi que d'un éventuel *feed-back* du KRG vers le GDRP.

Troisième partie : du modèle au circuit

Le chapitre 6 expose les différentes implantations possibles d'un système modèlisé par un KRG, au moyen de la méthodologie de *conception insensible aux latences* (LID). Dans une première section, nous rappelons les différentes implantations dynamiques et statiques de l'état de l'art pour les systèmes sans routage, et présentons nos propres variantes. Dans une deuxième section, nous nous appuyons sur les résultats de la section précédente pour proposer des implantations insensibles aux latences des KRG, aussi bien statiques et que dynamiques.

Quatrième partie : conclusion et annexes

Le chapitre 7 conclut ce manuscrit, et présente nos différentes idées sur des travaux futurs, en vue d'en améliorer les résultats.

L'annexe A regroupe un ensemble de définitions et propriétés mathématiques sur lesquelles s'appuient nos travaux. Dans une première section, nous rappelons des travaux sur la théorie *n*-synchrone, ainsi que les définitions et propriétés de nos opérateurs sur les séquences binaires. Dans une seconde section, nous rappelons les principaux résultats de la théorie des polyèdres, grâce auxquels nous établissons le lien entre KRG et modèle polyédrique. Enfin, la dernière section résume les définitions des graphes de permutation et d'intervalle, nécessaire aux permutations des jetons pour des consommations dans le désordre.

L'annexe B regroupe l'ensemble des preuves des lemmes, propositions et théorèmes du manuscrit.

Première partie

Modèles flot de données

Chapitre 2

Modèles existants

Nous sommes tous influencés par les outils que nous utilisons, en particulier : les formalismes que nous utilisons modèlent nos façons de penser, pour le meilleur ou pour le pire, et cela veut dire que nous devons être très prudents dans le choix de ce que nous apprenons et enseignons, il n'est pas vraiment possible de désapprendre.

Edsger Dijkstra, *EWD1305 : Answers to questions from students of Software Engineering*, 28 novembre 2000.

Sommaire

Les réseaux de processus, au sens le plus général du terme, sont depuis longtemps utilisés en tant que modèles de calcul formels pour la conception de systèmes embarqués dédiés, aussi bien logiciels que matériels. Les réseaux de Petri [187] ou les réseaux de Kahn [126] sont les représentants d'autant de grandes familles de modèles décrivant un système sous forme d'acteurs indépendants, échangeant données et/ou contrôle par le biais de canaux de communication. Ces modèles privilégient une vision des manipulations des données et de leur transport, abstraites sous forme de jetons, plutôt que de s'intéresser aux calculs effectifs et à leurs valeurs. Ils offrent des abstractions suffisamment simples de systèmes beaucoup plus complexes, permettant une analyse mathématique. Nous nous intéressons plus particulièrement aux modèles flots de données, pour leur capacité à représenter des applications à traitements intensifs et répétitifs de données, de type SIMD (*Single Instruction, Multiple Data*).

2.1 Principes

Une application SIMD peut être modélisée sous la forme d'un graphe, dont les sommets, ou *nœuds de calcul*, représentent les opérations effectuées sur un flot de données. Les arêtes reliant ces sommets correspondent à des mémoires tampons, ou *canaux* de communication ou *places*, selon le principe de file (ou *FIFO – First In, First Out*). De tels graphes constituent des modèles de calcul, dont la sémantique est parfaitement adaptée à la description de matériel avec de petites mémoires distribuées.

2.1.1 Définitions

L'un de ces modèles graphiques des plus connus, et sans doute des plus anciens, est le réseau de Petri, dont nous reprendrons certaines notations et terminologies par la suite.

Définition 2.1 (Réseau de Petri)**.** *Un réseau de Petri est un quadruplet* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M \rangle$ *tel que :*
- $\mathcal{N}$ *est un ensemble fini de nœuds,*
- $\mathcal{P}$ *est un ensemble fini de places,*
- $\mathcal{T}$ *est un ensemble d'arcs reliant places et nœuds, inclus ou égal à* $(\mathcal{N} \times \mathcal{P}) \cup (\mathcal{P} \times \mathcal{N})$,
- M *est une fonction attribuant un marquage initial à chaque place, telle que* $M : \mathcal{P} \to \mathbb{N}$.

Un réseau de Petri est donc un graphe orienté biparti. Pour toute place p, on note $\bullet p$ et $p\bullet$ les ensembles des nœuds précédents et suivants respectivement, définis par :

$$\bullet p \stackrel{\text{def}}{=} \{n \in \mathcal{N} \mid (n,p) \in \mathcal{T}\} \tag{2.1}$$

$$p\bullet \stackrel{\text{def}}{=} \{n \in \mathcal{N} \mid (p,n) \in \mathcal{T}\} \tag{2.2}$$

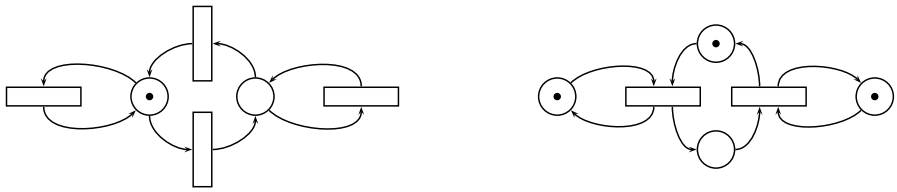
Aussi, pour tout nœud n, nous notons $\bullet n$ (resp. $n\bullet$) l'ensemble de ses entrées (resp. de ses sorties), tel que :

$$\bullet n \stackrel{\text{def}}{=} \{p \in \mathcal{P} \mid (p,n) \in \mathcal{T}\} \tag{2.3}$$

$$n\bullet \stackrel{\text{def}}{=} \{p \in \mathcal{P} \mid (n,p) \in \mathcal{T}\} \tag{2.4}$$

Un nœud n est dit *activé* ou *exécutable* dès lors que chacune de ses places en entrée dispose d'au moins une donnée. L'exécution, ou tirage, de n a pour effet de retirer un jeton de chaque entrée, et d'en ajouter un sur chaque sortie.

L'intérêt d'utiliser de tels modèles est de nous aider à décider, dans un cadre formel, de propriétés nécessaires à la synthèse de systèmes plus complexes :



(a) Réseau de Petri avec conflit : automate à deux états.

(b) Réseau de Petri sans conflit : graphe marqué.

FIG. 2.1: Exemples de réseaux de Petri avec et sans conflit.

Absence de blocage. Pour un réseau donné, et pour tout marquage atteignable par une séquence de tirages valides, existe-t-il toujours un nœud pouvant être exécuté ?

Vivacité. Un réseau est *quasi-vivant* si tous ses nœuds peuvent être exécutés au moins une fois à partir du marquage initial. Un réseau est *vivant* si chacun de ses nœuds peut s'exécuter à terme. Le réseau considéré vérifie-t-il ces propriétés ? Si oui, peut-il être ordonnancé périodiquement ?

Caractère borné. Un réseau est dit *borné* si, par toute séquence de tirages valides, le nombre de jetons dans chaque place est borné. Un réseau est dit *k-borné* si aucune de ses places ne contient plus de *k* jetons, par une quelconque séquence de tirages valides. Un réseau 1-borné est *sauf*. Ainsi, pour un réseau donné, peut-on dimensionner précisément ses mémoires ?

Cependant, les réseaux de Petri dans toute leur généralité ont une expressivité trop large pour nos attentes, qui englobe celle des automates finis. Dans un réseau de Petri, une place peut avoir plusieurs consommateurs. Le choix du tirage de l'un ou l'autre consommateur est exclusif, et introduit une notion de *conflit* : le comportement du réseau dépend alors de l'ordre dans lequel les jetons sont produits dans les places, et de *choix* internes, non-explicites, semblables à ceux définis par Hoare en CSP (*cf.* FIG. 2.1(a)). D'un point de vue logiciel, ce genre de comportements peut mener à des situations de compétition (*race conditions*) entre processus si aucun mécanisme d'exclusion mutuelle n'est établi.

2.1.2 Absence de conflit et propriétés induites

Nous nous intéressons donc à des modèles qui interdisent ce partage de places entre nœuds, de façon à obtenir un contrôle déterministe et *sans conflit*. Cette propriété est aussi connue sous le nom de *monotonicité* selon Kahn [126] ou de *confluence* dans des algèbres de processus tels que CCS [168]. Partant de ce principe, une place ne dispose plus que d'un unique producteur et d'un consommateur, est peut alors être vue comme un simple lien de communication point-à-point (*cf.* FIG. 2.1(b)). Ainsi, tout nœud activé le reste tant qu'il n'a pas été exécuté.

L'absence de conflit implique que le système n'a qu'un seul comportement possible, *modulo* l'ordre dans lequel sont exécutées les instructions indépendantes. L'essentiel des travaux sur la théorie de l'ordonnancement s'appliquent sur cette hypothèse (*cf.* Chrétienne *et al.* [60] et sous-section 2.1.3).

Autrement dit, dans un flot de données sans contrôle, la topologie du graphe et son marquage initial imposent un ordre partiel sur les tirages des nœuds. Toute séquence de tirages valide est donc une séquentialisation de cet ordre partiel. En ce point, les modèles que nous étudions rejoignent le principe des systèmes insensibles aux latences (*cf.* sous-section 2.2.3).

Si chaque nœud est étiqueté par un label représentant sa fonction, un parallèle peut être fait avec la théorie des traces [77] : un chemin entre deux nœuds induit une dépendance entre eux. Le complé-

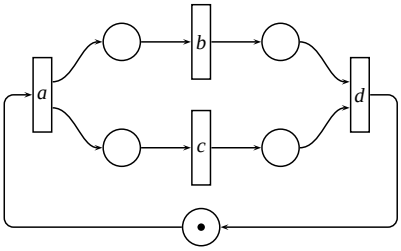


FIG. 2.2: Réseau de Petri sans conflit, où les tirages des nœuds *b* et *c* sont indépendants.

mentaire de la relation de dépendance est la relation d'indépendance, et les instants d'exécution de deux nœuds ne peuvent être permutés que si ces nœuds sont indépendants.

Exemple 2.2. Condidérons le réseau de Petri de la FIG. 2.2. Deux séquences de tirage franchissables sont *abcd* et *acbd*. Plus généralement, toutes les séquences de tirages franchissables appartiennent au langage défini par l'expression :

$$(a.(b.c|c.b).d)^* \tag{2.5}$$

Aussi, nous nous plaçons dans un cadre sans conflit pour la suite de ce manuscrit ; nous étudions des modèles adjoints d'un unique comportement partiellement ordonné.

2.1.3 Ordonnancement et hypothèses synchrones

Des *règles de tirage* particulières peuvent être définies pour que l'exécution du graphe suive un comportement donné. Par exemple, puisque le modèle permet la concurrence, et plutôt que de tirer les nœuds en séquence (ordonnancement *auto-temporisé*), on peut définir une notion de *pas* d'exécution, ou *instant logique*.

Temps logique

Le concept de temps *logique* est dû à Lamport [141], qui remarquait que «*le concept de temps est fondamental à notre façon de penser. Il est issu du concept encore plus basique d'ordre dans lequel les événements surviennent*». Dans ce sens, la modélisation de l'ordonnancement des tâches d'un programme peut être décorrélée du temps *réel* où elles sont exécutées. Il suffit pour cela que ce temps logique, vu comme un ensemble d'*instants*, et muni d'une relation de *précédence* entre passé et futur, réponde à un ensemble d'hypothèses fondamentales [228] :

1. La précédence entre instants est transitive.
2. La précédence est irréflexive : passé et futur sont strictement ordonnés.
3. Le temps est linéaire.
4. Le temps est éternel : à tout instant, il existe un futur.
5. Le temps est dense : pour toute paire d'instants distincts, il existe un instant intermédiaire.

Bien souvent, les instants sont assimilés à des entiers. Par exemple, dans un circuit électronique numérique, il n'existe pas de plus petite subdivision du temps que le cycle de l'horloge maîtresse. Dans ce cas, le dernier point est remplacé par :

5'. Le temps est discret.

Hypothèses synchrones

Le paradigme synchrone est le standard *de facto* pour la conception de circuits électroniques numériques, avec lesquels il entretient un lien sémantique fort. Il repose sur un ensemble d'hypothèses additionnelles :

1. Le temps est discret : c'est un ensemble d'instants, totalement ordonné par une horloge. Les instants peuvent être indicés sur $\mathbb{N}^*$;
2. À chaque instant, un module synchrone *réagit* : il effectue un ensemble d'actions simultanées, de durée nulle. Ces actions peuvent être partiellement ordonnées par des relations de *causalité* ;
3. Les communications entre modules se font par le biais de *signaux*, dont la propagation est instantanée vers tout autre module ;
4. Pour chaque réaction, tout signal est dans exactement un état et, le cas échéant, a une unique valeur : un module synchrone est réactif (au moins un état) et déterministe (au plus un état) ;
5. Pour chaque réaction, un module synchrone échantillonne l'ensemble de ses entrées, et émet un signal sur chacune de ses sorties. L'état, et le cas échéant la valeur, d'un signal d'entrant doit être connu au moment de l'échantillonnage.

Le respect de la causalité demande à ce qu'un système synchrone ne puisse avoir de cycle combinatoire : une boucle doit obligatoirement[1] contenir un élément mémoire, tel un registre ou une temporisation. Le modèle synchrone est dépourvu de contrôle : les structures conditionnelles sont expansées par «conversion de sélection» (*if-conversion*). Par exemple :

$$a = \begin{cases} c & \text{si } b \\ d & \text{sinon} \end{cases} \quad \Leftrightarrow \quad a = (b \wedge c) \vee (\overline{b} \wedge d)$$

D'un point de vue électronique, ces équations correspondent à un multiplexeur «2 vers 1» avec des entrées c et d, une sortie a, et une sélection b.

Ces hypothèses assurent une composition déterministe de modules concurrents. Elles apportent une sémantique forte au modèle, et offrent un cadre formel dans lequel peuvent être développées de nombreuses méthodes d'analyse, de compilation et de vérification. De nombreux langages se basent sur les hypothèses synchrones, tel qu'Esterel [193], ou les étendent. Nous renvoyons le lecteur à Nebut [179] et Potop *et al.* [192] pour un état de l'art et une comparaison des principaux langages synchrones.

Approche *n*-synchrone

La plupart des langages synchrones permettent de définir des horloges à partir de n'importe quelle expression booléenne. Bien que cela offre une grande expressivité pour la description des systèmes, leur composition peut s'avérer contraignante : il est nécessaire de prouver l'égalité des horloges d'activation pour tout couple producteur-consommateur.

L'approche *n*-synchrone [62, 63] consiste à relâcher ces contraintes de composition synchrone : deux processus dont les horloges ne sont pas synchrones peuvent néanmoins communiquer via des mémoires tampons de taille n. Il suffit de s'assurer (i) (*précédence*) la production d'une donnée précède sa production, et (ii) (*synchronisabilité*) les horloges ont globalement la même fréquence. Ces

[1] Plus précisément, seule une boucle *logique* entraînant la divergence d'un signal sortant demande à être cassée [202]. Pour s'en convaincre, une bascule est bien constituée d'une boucle entre deux portes *non-et* ou *non-ou*, et son comportement est réactif et déterministe *à condition* de prévenir le cas particulier : $(1,1)$ pour une bascule RS, $(0,0)$ pour une bascule $\overline{\text{R}}\overline{\text{S}}$.

propriétés mènent à la définition de règles de typage, à partir desquelles peuvent être automatiquement inférées les tailles des mémoires nécessaires aux communications. Ce principe était initialement appliqué à des horloges représentées par des séquences binaires *k*-périodiques (*cf.* annexe A.1). Dans sa thèse, Plateau prouve qu'il peut être étendu à des horloges abstraites sous la forme d'une *enveloppe*, où seuls la fréquence asymptotique et les décalages minimum et maximum sont connus [190].

Lucy-*n* est un langage de programmation *n*-synchrone où les horloges peuvent être définies aussi bien sous la forme de séquences binaires *k*-périodiques que d'enveloppes [158].

Approche polychrone

L'approche polychrone enrichit les précédentes hypothèses au cas où le temps est dirigé par non pas une seule, mais plusieurs horloges. Le temps correspond alors à un ensemble d'instants partiellement ordonnés, tel que chaque horloge n'établisse un ordre total que sur un sous-ensemble d'instants. Les instants de chaque horloges sont *a priori* indépendants, et ne sont pas astreints à coïncider. Des opérateurs synchrones ou asynchrones imposent néanmoins des relations entre horloges. Au final, un modèle polychrone consiste en un ordre partiel global, construit sur les ordres totaux locaux des horloges. Il permet donc de représenter des systèmes de type GALS.

Cette approche est notamment promue par le métamodèle *tagged-signal*, visant à étudier les comportements de processus concurrents, et en particulier leurs compositions. Des langages tels que Signal [24] et Esterel multi-horloges [26] se basent sur ce genre de notions. Leurs concepts sont généralisés par le langage de spécification CCSL (*Clock Constraints Specification Language*) [8, 9].

Pour un réseau de processus flot de données, la notion de calcul *aussitôt que possible* (ASAP), calque une exécution en flux tendu, où tous les nœuds de calcul sont exécutés simultanément, aussitôt qu'ils sont activés. Une telle règle de tirage associe une sémantique *polychrone* au modèle. Le modèle est *synchrone* lorsque tous les nœuds s'exécutent à chaque instant : c'est le cas si, à chaque instant, tout nœud consomme un jeton sur chaque entrée, et produit un jeton sur chaque sortie.

2.1.4 Modèles sans conflit

Une première famille de modèles, connus sous le nom de *flots de données purs*, exclut toute forme de contrôle. Ces modèles sont particulièrement adaptés pour exprimer des ordonnancements statiques et réguliers. Par ailleurs, un ordonnancement borné demande à ce que tout nœud activé soit ultimement exécuté. Ceci se traduit par la propriété d'*équité*[2] (*fairness*) : pour tout couple de nœuds, le nombre de tirages de l'un sans que l'autre ne soit exécuté est borné.

D'autres familles de modèles, plus élaborés, prennent en compte le contrôle propre aux nœuds. Ce contrôle peut être statique ou dynamique, abstrait sous la forme d'un oracle interne au nœud ou imposé par des conditions explicites.

Dans la suite de ce chapitre, nous présentons différents réseaux de processus sans conflit, pour la modélisation de flots de données, et en discutons les avantages et les inconvénients.

[2] De nombreuses et différentes définitions de l'équité ont été proposées dans la littérature. Se referer à Murata [174] pour une présentation plus complète.

2.2 Flots de données purs

2.2.1 Graphe marqué : définition et propriétés

Les graphes marqués (*marked graphs*), introduits par Genrich [97] puis étudiés par Commoner *et al.* [66], sont une sous-famille des réseaux de Petri. Ils sont également présents dans la littérature sous le nom de graphes d'événements (*event graphs*).

Définition 2.3 (Graphe marqué)**.** *Un réseau de Petri $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M \rangle$ est un graphe marqué si et seulement si chacune de ses places a un unique producteur et un unique consommateur :*

$$\forall\, p \in \mathcal{P},\ |\bullet p| = |p\bullet| = 1 \tag{2.6}$$

Un graphe marqué est un modèle permettant la concurrence, mais sans conflit. Dorénavant, nous notons $\bullet p$ et $p\bullet$ les éléments (nœuds) plutôt que les singletons. Par exemple, si (n, p) et $\left(p, n'\right)$ appartiennent à $\mathcal{T}$, alors $\bullet p = n$ et $p\bullet = n'$. n est le producteur de p, et n' est son consommateur.

Nous rappelons quelques résultats fondamentaux sur les graphes marqués contenant des composantes fortement connexes [66].

Lemme 2.4. *Pour tout cycle simple d'un graphe marqué, le nombre de jetons est invariant par tirage des nœuds.*

Théorème 2.5. *Un marquage est vivant si et seulement si le nombre de jetons sur chaque cycle est strictement positif.*

Théorème 2.6. *Un marquage vivant est sauf si et seulement si chaque place du graphe appartient à un cycle dont le nombre de jetons est unitaire.*

Théorème 2.7. *S'il existe une séquence de tirages pour un graphe connexe, et que cette séquence ramène au marquage initial, alors tous les nœuds ont été tirés un même nombre de fois.*

Théorème 2.8. *Soit M un marquage vivant. Il existe une séquence de tirages menant de M à lui-même, dans laquelle chaque nœud est exécuté exactement une fois.*

Théorème 2.9. *Dans un graphe fortement connexe, si un marquage M' peut produire M par une séquence franchissable de tirages, alors M peut produire M'.*

Les théorèmes 2.7 à 2.9 donnent les premiers éléments quant à l'ordonnancement périodique des graphes marqués. Cet aspect fut étudié dans les détails, en particulier, dans Carlier-Chrétienne [47], puis dans Baccelli *et al.* [14].

Dans le cas général, effectuer des traitement sur des données, ou les transporter d'un nœud de calcul à un autre, n'est pas instantané : cela prend du temps. Des *latences* de calcul et de communication furent introduites dans les réseaux de Petri par Ramchandani [200]. Ces latences correspondent à un temps logique et discret, représenté par des entiers naturels.

Exemple 2.10. Une analogie simple est celle de la chaîne d'assemblage dans un atelier : les nœuds de calculs sont des machines, les places sont des tapis roulants, et les jetons sont des pièces. Une machine prend *x instants* (l'unité de temps est arbitraire) pour assembler une pièce, puis cette pièce met *y instants* pour transiter jusqu'à la prochaine machine sur le tapis roulant, et ainsi de suite.

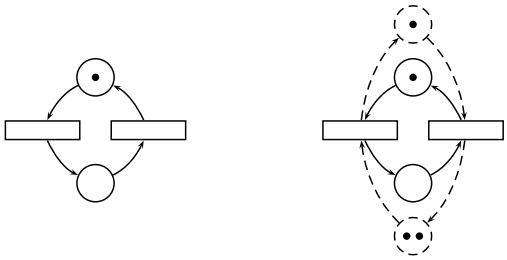


FIG. 2.3: Un graphe marqué de capacité deux (*gauche*) et son équivalent sans capacité (*droite*).

Aussi, nos modèles doivent tenir compte du fait que les systèmes ont, dans la pratique, des *capacités* de stockage limitées. Cette notion rejoint celle du caractère borné d'un graphe, dans le sens où l'on tient à s'assurer que le nombre de jetons stockés dans une place à un instant donné ne peut pas dépasser cette capacité. Il serait irréaliste de vouloir produire un système où des données s'accumuleraient indéfiniment.

Exemple 2.11. Maintenant, la chaîne d'assemblage est divisée en deux ateliers distants, qui échangent leurs pièces via un entrepôt. Outre les contraintes sur sa propre chaîne de montage, le producteur doit veiller à respecter une nouvelle contrainte : le volume de stockage de l'entrepôt est limité. Si l'entrepôt est plein, le producteur doit attendre que le consommateur libère de la place avant de pouvoir continuer à y envoyer de nouvelles pièces.

Compte tenu des propriétés de latence et de capacité, la définition 2.3 est étendue ; les définitions correspondantes varient selon les auteurs, nous donnons la définition 2.12 pour coller à nos notations.

Définition 2.12 (Graphe marqué temporisé avec capacités)**.** *Un graphe marqué temporisé est un sextuplé* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, L, K \rangle$ *tel que :*

- $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M \rangle$ *est un graphe marqué,*
- L *est la fonction de latence, associant un poids à chaque nœud et place, telle que* $L : \mathcal{N} \cup \mathcal{P} \to \mathbb{N}$,
- K *est la fonction de capacité, associant à chaque place le nombre maximum de jetons qu'elle peut contenir, telle que* $K : \mathcal{P} \to \mathbb{N} \cup \{+\infty\}$.

Un graphe avec capacités finies peut cependant être transposé en un graphe équivalent sans capacité[3], tel que : $\forall\, p \in \mathcal{P}, K(p) = +\infty$. Pour se faire, on associe à chaque place une place complémentaire [7, 14] : pour toute place p de capacité finie, on ajoute au graphe une place complémentaire $\overline{p}$, avec $\bullet p = \overline{p}\bullet$ et $p\bullet = \bullet\overline{p}$ (*cf.* FIG. 2.3). Par la suite, pour tout ensemble de places $\mathcal{P}$, nous notons $\overline{\mathcal{P}}$ l'ensemble de ses places complémentaires.

Définition 2.13 (Graphe complémenté)**.** *Soit* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, L, K \rangle$ *un graphe marqué temporisé connexe de capacité finie. Le* graphe complémenté $\langle \mathcal{N}', \mathcal{P}', \mathcal{T}', M', L', K' \rangle$ *est son équivalent avec places com-*

[3] Ou plus précisément, à capacités infinies, telles qu'elles ne soient plus un facteur limitant.

plémentaires et de capacités infinies, tel que :

$$\mathcal{N}' \stackrel{\text{def}}{=} \mathcal{N} \tag{2.7}$$

$$\mathcal{P}' \stackrel{\text{def}}{=} \mathcal{P} \cup \overline{\mathcal{P}} \tag{2.8}$$

$$\mathcal{T}' \stackrel{\text{def}}{=} \mathcal{T} \cup \{(n,\overline{p}) \mid n \in \mathcal{N}, p \in \mathcal{P}, (p,n) \in \mathcal{T}\} \cup \{(\overline{p},n) \mid n \in \mathcal{N}, p \in \mathcal{P}, (n,p) \in \mathcal{T}\} \tag{2.9}$$

$$\forall p \in \mathcal{P},\ M'(p) = M(p) \tag{2.10}$$

$$M'(\overline{p}) = K(p) - M(p) \tag{2.11}$$

$$L'(p) = L'(\overline{p}) = L(p) \tag{2.12}$$

$$K'(p) = +\infty \tag{2.13}$$

Exemple 2.14. Considérons l'exemple en FIG. 2.3 : un graphe de capacité deux a pour équivalent un graphe sans capacité, tel que le complément d'une place contenant initialement un jeton sera de $2-1=1$ jeton, et qu'une place vide sera complémentée par une place contenant $2-0=2$ jetons.

Exemple 2.15. Considérons maintenant un graphe dont le marquage initial sature les capacités des places : $\forall p \in \mathcal{P},\ M(p) = 2$. Son équivalent sans capacité contient un cycle ne contenant aucun jeton, violant ainsi la condition nécessaire et suffisante du théorème 2.5 pour que le graphe soit vivant. Autrement dit, une saturation (ou verrou vivant) d'un graphe avec capacités implique une famine (ou verrou mortel) dans son équivalent sans capacité.

Comme mentionné précédemment, les latences représentent une notion de temps discret et logique. Quelque soit la règle de tirage choisie, l'ordonnancement d'un nœud de calcul sous la forme d'une séquence binaire ultimement périodique (*cf.* section A.1). Chaque lettre de la séquence correspond à un instant, et vaut «1» si le nœud est tiré (il commence un nouveau traitement), ou «0» sinon. À partir de là découle une notion de *débit*.

Définition 2.16 (Débit d'un nœud)**.** *Soit n un nœud d'un graphe marqué temporisé. Pour une règle de tirage donnée, son débit est égal à la limite, lorsque le nombre d'instants écoulés tend vers l'infini, au ratio entre le nombre de ses tirages et ce nombre d'instants écoulés.*

Le débit d'un nœud est donc égal au débit de la séquence binaire qui représente son ordonnancement. Par le lemme 2.4 et le théorème 2.7, nous savons que pour une séquence de tirage suffisamment longue, les débits de tous les nœuds d'un circuit tendent à être égaux. D'où les définitions suivantes :

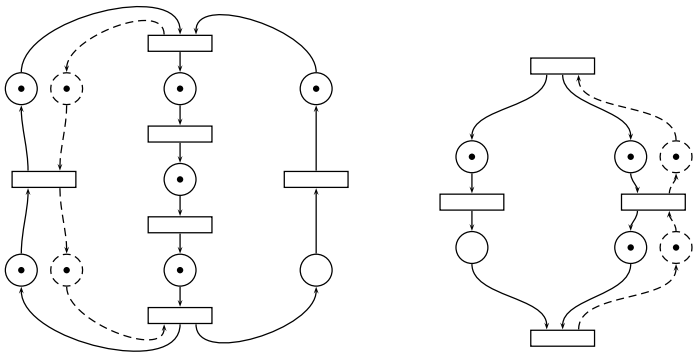
Définition 2.17 (Débit d'un circuit)**.** *Soit c un circuit d'un graphe marqué temporisé. Son débit* $\Theta(c)$ *est le ratio entre le nombre d'exécutions de chacun de ses nœuds et le temps écoulé.*

Théorème 2.18 (Débit d'un graphe)**.** *Le débit d'un graphe marqué temporisé est la fréquence d'exécution maximale de ses nœuds. Le débit d'un graphe fortement connexe est égal au débit du plus lent de ses circuits. Le débit d'un graphe acyclique est* 1.

Définition 2.19 (Circuit critique)**.** *Un circuit est dit critique si son débit est égal au débit du graphe.*

2.2.2 Graphe marqué : capacité et débit

Considérons désormais une règle de tirage ASAP, afin de nous placer dans les mêmes conditions qu'un système insensible aux latences (*cf.* sous-section 2.2.3). Afin de maximiser les performances du système modélisé, nous cherchons à en maximiser le débit.



(a) Graphe cyclique. (b) Graphe acyclique.

FIG. 2.4: Exemples de graphes dont le débit est limité à $3/4$ à cause des places 2-bornées. Les places en traits pleins appartiennent au graphe avec capacités finies ; nous ne dessinons, en pointillés, que les places complémentaires les plus significatives. Les latences des nœuds sont nulles, et les latences des places sont unitaires.

Cependant, comme nous allons le voir au travers des remarques suivantes, il existe un lien fort entre le débit maximum atteignable et les capacités des places. Notons par ailleurs que la propriété d'*acyclicité* perd de son sens dans un graphe connexe avec capacités, dans le sens où il est équivalent à un graphe fortement connexe : son graphe complémenté. Nous cherchons alors à maximiser le débit atteignable, c'est-à-dire le débit du graphe lorsque les capacités sont assez grandes pour ne plus être un facteur limitant (*suffisantes*), tout en minimisant la capacité totale (*nécessaire*).

Remarque 2.20. Certains travaux [51, 56, 156, 217] font le lien avec le problème du *cycle moyen maximum* [14, 47] pour calculer le débit d'un graphe marqué temporisé. Le débit $\Theta(c)$ d'un circuit c est défini tel que :

$$\Theta(c) = \frac{|\mathcal{N}_c|}{|\mathcal{N}_c| + \sum\limits_{p \in \mathcal{P}_c} L(p)} \tag{2.14}$$

où $\mathcal{N}_c$ et $\mathcal{P}_c$ sont respectivement les sous-ensembles de nœuds et places appartenant au circuit c. Les auteurs font l'hypothèse que les latences des nœuds sont unitaires ; plus généralement, on peut remplacer dans leur formule le $\mathcal{N}_c$ du dénominateur par $\sum_{n \in \mathcal{N}_c} L(n)$.

L'équation (2.14) signifie que le débit est défini comme étant égal au nombre de nœuds du circuit, divisé par la somme des latences. Cette formulation ne tient pas compte du marquage initial : elle n'est consistante que si l'on suppose un système contenant autant de jetons initiaux que de places. C'est un cas particulier où chaque nœud dispose d'une valeur initiale sur chacune de ses entrées, comme par exemple dans Dasdan-Gupta [74]. Ce n'est donc pas le cas général.

Remarque 2.21. Considérons deux graphes marqués temporisés, de topologie identique, l'un avec des capacités finies, l'autre avec des capacités infinies. Comme remarqué par Millo [166], le graphe à capacités bornées peut s'exécuter avec un débit inférieur à celui d'une version à capacités infinies.

Nous donnons en FIG. 2.4(a) un exemple de graphe 2-borné. Toutes les places appartiennent au graphe à capacités finies, tandis que seules les places en traits pleins appartiennent au graphe à capacités infinies. Les places complémentaires sont donc en pointillés ; nous n'en représentons que

deux, les plus significatives dans cet exemple. Nous supposons que les latences des places sont nulles, et que les temps de calcul sont unitaires.

Le graphe avec capacités infinies contient deux cycles : l'un de débit 1 (à gauche), l'autre de débit $4/5$ (à droite). Le débit de ce graphe est donc égal à $4/5$. Cependant, les places complémentaires introduisent de nouveaux cycles dans le graphe avec capacités finies : en particulier un cycle de débit $3/4$. Ainsi, le débit du graphe avec capacités finies est égal à $3/4$.

Remarque 2.22. Le débit d'un graphe acyclique *avec capacités finies* peut également être inférieur à 1. La FIG. 2.4(b) y présente le rôle du marquage initial. Sur cet exemple, les deux branches ont un marquage initial déséquilibré, avec une *bulle* dans la branche gauche. Lorsque nous prenons en compte les places complémentaires, nous créons un circuit de débit $3/4$.

Seuls les arbres avec des capacités supérieures ou égales à deux échappent à ce phénomène ; les places complémentaires n'y introduisent pas d'autre circuit simple que ceux formés par une place et son complémentaire.

Maintenant que nous avons pris en compte les contraintes induites par des capacités finies, nous pouvons donner l'expression du débit atteignable d'un graphe.

Théorème 2.23 (Débit atteignable)**.** *Soit $g = \langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, L, K \rangle$ un graphe marqué temporisé connexe avec capacités finies, et $g' = \langle \mathcal{N}', \mathcal{P}', \mathcal{T}', M', L', K' \rangle$ son graphe complémenté. Le débit maximum atteignable $\Theta(g)$ de g est :*

$$\Theta(g) = \min_{c \subseteq g'} \left(\frac{\sum_{p \in \mathcal{P}'_c} M'(p)}{\sum_{n \in \mathcal{N}'_c} L'(n) + \sum_{p \in \mathcal{P}'_c} L'(p)}, 1 \right) \tag{2.15}$$

où $c \subseteq g'$ représente, par abus de notation, tout circuit c du graphe g'.

Le débit maximum du graphe est un cas particulier du débit atteignable, obtenu en supposant que les capacités du graphe sont infinies : les circuits introduits par le graphe complémentaire ne peuvent pas faire chuter le débit global. Une fois le débit maximum connu, nous minimisons la somme des capacités des places nécessaires pour l'atteindre. Nous calculons alors une fonction capacité K, donnée par le programme linéaire en nombres entiers (PLNE) suivant :

$$\text{minimiser} \sum_{p \in \mathcal{P}} K(p) \tag{2.16}$$

avec, pour tout circuit c de g', tel que c contienne au moins une place complémentaire, des contraintes de la forme :

$$\sum_{p \in \mathcal{P}'_c} M'(p) \times \left(\sum_{n \in \mathcal{N}'_\gamma} L'(n) + \sum_{p \in \mathcal{P}'_\gamma} L'(p) \right) - \sum_{p \in \mathcal{P}'_\gamma} M'(p) \times \left(\sum_{n \in \mathcal{N}'_c} L'(n) + \sum_{p \in \mathcal{P}'_c} L'(p) \right) \geq 0 \tag{2.17}$$

où γ est le circuit critique de g sans prendre en compte ses capacités, et pour tout p de $\mathcal{P}$:

$$K(p) = M'(p) + M'(\overline{p}) \tag{2.18}$$

$$K(p) \geq 0 \tag{2.19}$$

Remarque 2.24. Les inéquations précédentes peuvent être modifiées en remplaçant le débit optimal par un débit «cible» Θ_{cible}, inférieur ou égal. Dans ce cas, $\sum_{p \in \mathcal{P}'_\gamma} M'(p)$ et $\sum_{n \in \mathcal{N}'_\gamma} L'(n) + \sum_{p \in \mathcal{P}'_\gamma} L'(p)$ sont à remplacer par k et p respectivement, tels que $\Theta_{\text{cible}} = \frac{k}{p} \geq \Theta(\gamma)$.

Entrées : un graphe marqué temporisé avec capacités $g = \langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, L, K \rangle$.
Sorties : le dimensionnement des mémoires de g.

Étape 1 : appliquer au graphe non-complémenté g une variante de l'algorithme des plus courts chemins de Johnson [124] :
pour tout nœud n de $\mathcal{N}$ **faire**
    rechercher le plus court circuit passant par n, s'il existe
    la métrique de poids est le ratio entre les marquages initiaux et les latences du chemin
fin pour
Étape 2 : calculer le débit potentiel du graphe
si aucun circuit n'est trouvé **alors**
    débit du graphe = 1
sinon
    débit du graphe = minimum des débits des circuits
fin si
Étape 3 : construire le graphe complémenté g'
Étape 4 : énumérer tous les circuits de g' contenant au moins un arc complémentaire avec l'algorithme de Johnson [123]. Afin d'optimiser cette recherche, nous effectuons un parcours en profondeur qui vérifie, lorsqu'un circuit est trouvé, qu'il contient au moins une place complémentaire ; les autres circuits sont ignorés
Étape 5 : construction des contraintes :
pour tout circuit ainsi énuméré **faire**
    construire une contrainte de la forme de l'inéquation (2.17)
    L' et M' doivent vérifier les équations (2.10), (2.11) et (2.12)
fin pour
Étape 6 : résoudre le PLNE (2.16) sous les contraintes de l'étape 5
retourner les résultats

Algorithme 2.1: Calcul les capacités minimales des places

Remarque 2.25. Une autre variante consiste à borner les capacités de certaines places, ce qui se traduit par des contraintes supplémentaires de la forme $K(p) \leq K_{max}(p)$.

Enfin, l'algorithme 2.1 permet de calculer les capacités minimales des places, tout en permettant d'atteindre le débit maximum. Il construit le système d'équations du PLNE précédemment énoncé, et le résoud. Il est semblable à notre précédent algorithme de Boucaron *et al.* [36]. Les complexités temporelles des différentes étapes de l'algorithme sont les suivantes :

- Étape 1 : $O\left(|\mathcal{P}|^2 \times \log_2 |\mathcal{P}| + |\mathcal{N}| \times |\mathcal{P}|\right)$
- Étape 2 : $O(|\mathcal{N}|)$
- Étape 3 : $O(|\mathcal{N}| + |\mathcal{P}|)$
- Étape 4 : $O\left(\left(|\mathcal{N}'| + |\mathcal{P}'|\right) \times (c+1)\right)$, où c est le nombre de circuits élémentaires dans g'
- Étape 5 : $O(|\mathcal{P}| \times c)$
- Étape 6 : NP-complet, dépend de l'heuristique utilisée par le solveur

2.2.3 Graphe marqué : conception insensible aux latences

La méthodologie de conception dite «insensible aux latences» (*latency-insensitive design – LID*) [49], aussi connue sous le nom d'élastique synchrone (*synchronous elastic flow* – SELF) [70], permet

de générer systématiquement un équivalent fonctionnel à une spécification synchrone, tolérant aux latences d'interconnexion. Elle sépare la considération de la fonctionnalité des modules et celle de leur implantation.

Protocole insensible aux latences

Le principe du LID repose sur le protocole insensible aux latences (*latency-insensitive protocol* – LIP) [50] ; il garantit la fonctionnalité d'un système composé de modules localement synchrones qui, du fait des latences de communications, ne respecterait pas les hypothèses synchrones au niveau global.

Le LIP se base sur le concept de processus *patient* : un processus synchrone dont la fonctionnalité ne dépend que de la valeur et de l'ordre de ses données en entrée, et non de leurs instants d'arrivée. Autrement dit, une séquence d'entrée implique toujours la même séquence de sortie, *modulo* des distances temporelles différentes. La composition de système patient est elle-même un système patient.

Cependant, la patience d'un processus est une contrainte forte : par définition, un module synchrone échantillonne toutes ses entrées et émet sur toutes ses sorties à chaque instant. Par conséquent, la plupart des modules synchrones ne sont pas patients. De façon à assurer ce pré-requis, le LIP utilise deux composants :

La coquille. Un module synchrone, alors appelé *perle*, est encapsulé dans une *coquille*. Le rôle de la coquille est double : la synchronisation des entrées, la perle étant exécutée dès lors que toutes ses entrées disposent d'une donnée ; et la propagation en sortie, en s'assurant que le système en aval est prêt à recevoir la donnée produite.

La station relai. Les canaux de communication trop longs ne peuvent être traversés par un signal en un cycle d'horloge, comme mentionné ci-dessus. Ils sont alors segmentés en tronçons plus courts, séparés par des *stations relais*. Une station relai est un processus patient permettant de stocker des données d'un cycle d'horloge à l'autre. Elle est placée à l'extrémité d'un tronçon de façon à ce que l'émetteur et le récepteur sur ce tronçon ne soient mutuellement distants que d'un cycle d'horloge. La capacité minimale d'une station relai est de deux places, pour ne pas dégrader les performances du système.

Un mécanisme de *contre-pression* permet de contrôler les flux dynamiquement : lorsqu'une station relai est pleine, ou qu'une coquille ne peut recevoir de données sur l'une de ses entrées, celle-ci émet vers l'amont un signal d'arrêt. Ce signal a pour but d'indiquer à l'émetteur que le récepteur n'est pas en mesure de traiter une nouvelle donnée dans l'instant. La contre-pression est alors rétro-propagée dans le pipeline pour arrêter ses étages jusqu'à la fin de la congestion. Notons que Harris avait proposé, dès 2000, une idée de temporisation de flot de données, similaire à une propagation de contre-pression [117]. Ce document, s'attachant essentiellement aux aspects de l'implantation, y détaille l'architecture de blocs comparables à des stations relais.

En définitive, le protocole insensible aux latences repose sur deux hypothèses. La principale est que les perles peuvent être arrêtées en un seul cycle d'horloge. La seconde est que l'on travaille en latences entières, ce qui est toujours possible en retardant les signaux jusqu'au prochain front d'horloge.

La méthodologie est détaillée par Carloni *et al.* [49] : le développeur conçoit un système synchrone, composé de différents modules, reliés par des canaux de communication point-à-point. Ces modules sont encapsulés par des coquilles, faisant interface avec les stations relais. Les latences d'interconnexion sont connues après placement-routage ; les canaux sont segmentés itérativement pour y insérer le nombre adéquat de stations relais, afin de respecter les contraintes de temps. En effet, le

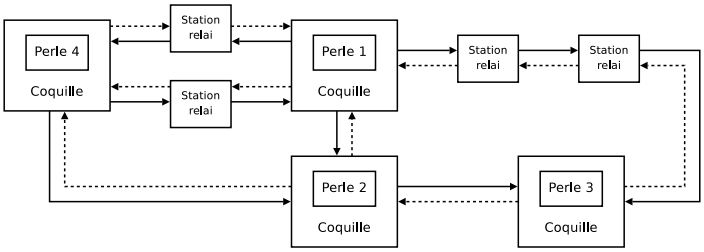


FIG. 2.5: Exemple de système insensible aux latences. Les flèches en traits pleins représentent les canaux point-à-point, tandis que celles en pointillés correspondent au protocole contre-pression.

```
main module Test :
  ...
  signal S2 : integer in
    run A/SousModule [S1/I, S2/O]
    ||
    run B/SousModule [S2/I, S3/O]
  end signal
end module
```

```
module SousModule :
  input I : integer;
  output O : integer;

  sustain ?O <= f(?I) if I
end module
```

FIG. 2.6: Exemple de programme Esterel où l'évaluation d'une instance *B* dépend d'une autre instance *A*.

placement des blocs n'est pas connu *a priori*, et d'autant moins les longueurs et délais des interconnexions, qui doivent être estimés et progressivement raffinés. Un exemple de système insensible aux latences est représenté en FIG. 2.5.

Modélisation par un graphe marqué

Par ailleurs, un système synchrone peut être modélisé par un graphe marqué [33, 39, 40, 166]. Plus précisément, un LIS est modélisé par un graphe marqué temporisé à capacités bornées. La règle de tirage d'un nœud de calcul correspond au comportement d'une perle et sa coquille : une donnée est consommée sur chaque entrée, et une donnée est produite sur chaque sortie. Une station relai contenant deux registres est assimilable à une place de capacité égale à deux.

Nous attirons toutefois l'attention du lecteur sur le fait que chaque place ne contient pas nécessairement de jeton initial. Considérons le code Esterel de la FIG. 2.6, décrivant un système composé de deux modules synchrones *A* et *B*. Ces modules sont déclarés comme s'exécutant en parallèle, mais la dépendance introduite par le signal S_2 entre la sortie de *A* et l'entrée de *B* force la séquence. L'évaluation de *B* dépend, dès le premier instant, de celle de *A*. Même si la *présence* d'un signal valué peut toujours être testée, sa *valeur* ne peut être lue que lorsqu'il est présent. Dans un graphe marqué, un jeton représente une donnée ; attribuer des jetons initiaux est sémantiquement équivalent à «le registre contient initialement une valeur valide». La conception insensible aux latences ne permet pas de faire de telles hypothèses. Il n'est pas toujours possible d'initialiser un étage de pipeline si ses calculs dépendent de données produites en amont (*e.g.* un décodeur vidéo ou un algorithme de cryptographie) ; le LIP ne peut introduire plus de jetons que spécifiés par le concepteur. Ainsi, dans le graphe marqué correspondant à ce programme Esterel (*cf.* FIG. 2.7), la place entre *A* et *B* a un marquage vide.

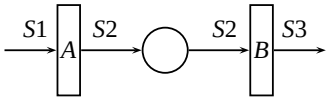


FIG. 2.7: Exemple de place sans marquage initial.

Le théorème 2.23 nous permet de calculer le débit maximum atteignable par un LIS, compte tenu des capacités de ses stations relais et de ses latences. D'autre part, le calcul des capacités minimales des stations relais sur un canal, tout en maximisant le débit, se fait au moyen du programme linéaire en nombres entiers (2.16). Pour cela, nous ajoutons des contraintes supplémentaires pour toute place p de la forme :

$$K(p) \geq 2 \times L(p) \tag{2.20}$$

2.2.4 Graphes flots de données synchrones

Lee et Messerschmitt proposent une extension des graphes marqués en graphes flots de données synchrones (*synchronous data flow – SDF*) [144], également connus dans la littérature sous le nom de graphes d'évènements pondérés (*weighted event graphs*). En SDF, les nœuds de calcul ne se limitent plus à consommer ou à produire qu'un seul jeton par place et par tirage. Des poids entiers sont associés aux entrées et sorties des places. Ces poids représentent un nombre explicite de jetons qui sont produits (resp. consommés) dans la place lorsque son producteur (resp. son consommateur) est exécuté.

Définition 2.26 (Graphe flot de données synchrone)**.** *Un graphe* flot de données synchrone *est un quintuplet* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, W \rangle$ *tel que :*

- $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M \rangle$ *est un graphe marqué,*
- W *est une fonction de poids, attribuant à chaque place* p *le nombre de jetons produits et consommés par tirage de* $\bullet p$ *et* $p\bullet$ *respectivement, telle que* $W : \mathcal{P} \rightarrow \mathbb{N}^2$.

Les définitions de l'activation et de l'exécution d'un nœud doivent être adaptées en conséquence. Les questions sur les bornes et l'absence de blocage sont en quelque sorte plus compliquées à résoudre pour les graphes SDF que les graphes marqués. Un important résultat de Lee-Messerschmitt [143] fournit un critère pour vérifier le caractère borné, en résolvant les équations dites *de balance* ; elles reposent sur le principe qu'au sein d'un flot de données, la production doit compenser la consommation. Plus précisément, les équations de balances permettent de vérifier, non pas si le graphe SDF est borné dans l'absolu, mais s'il peut être ordonnancé de façon à borner les capacités de ses places. À partir d'un graphe SDF, il est alors possible de construire un système d'équations linéaires de la forme :

$$\forall p \in \mathcal{P},\ \ W(p) = (i_p, o_p),\ \ x \times i_p = y \times o_p \tag{2.21}$$

où x_n et $x_{n'}$, les deux variables, correspondent respectivement au nombre de tirages du producteur n et du consommateur n' de p, de telle façon que le nombre de jetons produits par l'un soit exactement celui consommé par l'autre. Il existe une solution triviale, telle que pour tout n, $x_n = 0$:

- S'il n'existe aucune autre solution, alors il est impossible de trouver un ordonnancement tel que le graphe soit borné : dans ce cas, quelque soit l'ordonnancement choisi, il y aura accumulation de jetons dans une place.
- Sinon, il existe au moins un ordonnancement périodique où x_n et $x_{n'}$ correspondent aux nombres de tirages de n et n' par période.

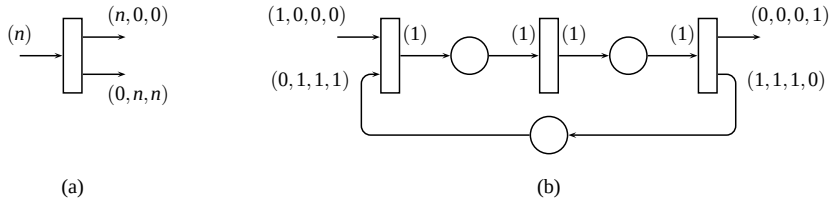


FIG. 2.8: Exemples de routage en CSDF : (a) branchement, (b) boucle.

Différentes techniques ont alors été développées pour optimiser le débit, les capacités des places, ou encore la longueur de la période [30, 103].

Une première solution, pour vérifier l'absence de blocage et la vivacité d'un graphe SDF borné, consiste à simuler son exécution jusqu'à revenir à l'état initial (sur une période, pour une ordonnancement périodique). Des solutions plus élaborées ont été proposées dans Marchetti-Kordon [159, 160].

Le modèle SDF fut par la suite étendu au cas des données multi-dimensionnelles [175, 176]. Notons que les graphes de calcul (*computation graphs*) [127] sont une généralisation des graphes SDF, où chaque place se voit attribuer un *seuil* d'activation ; il représente le nombre de jetons nécessaires à l'activation du nœud consommateur, obligatoirement supérieur ou égal à sa consommation.

2.3 Flots de données à contrôle statique

Les graphes exclusivement flots de données, dépourvus de contrôle et de routage comme jusqu'à présent, peuvent être étudiés statiquement. Le prix à payer est une certaine restriction quant aux classes de programmes à modéliser. Afin de relâcher cette contrainte, nous introduisons désormais du contrôle statique dans ces modèles. Les jetons y sont aiguillés selon des conditions de routage connues à la compilation.

2.3.1 Graphes flots de données cyclo-statiques

Les graphes flots de données cyclo-statiques (*cyclo-static dataflow – CSDF*) [31, 81] sont de tels modèles. Ce sont des extensions des graphes SDF, tels que les poids associés à la production et la consommation de certains de leurs nœuds ne sont plus de simples entiers, mais des listes d'entiers : ainsi les poids varient de façon cyclique et périodique. Cet ajout au modèle introduit une notion de *modes* en SDF, ici appelés *phases*. La plupart des résultats sur les équations de balance de SDF peuvent être transposés en CSDF.

Définition 2.27 (Flot de données cyclo-statique)**.** *Un graphe* flot de données cyclo-statique *est un quintuplet* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, W \rangle$ *tel que :*

- $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M \rangle$ *est un graphe marqué.*
- W *est une fonction associant à chaque nœud* $\bullet p$ *et* $p\bullet$*, pour toute place* p *de* $\mathcal{P}$*, une liste circulaire de poids dans* $\mathbb{N}$*.*

Ces poids variables permettent de modéliser du routage de données avec CSDF, bien qu'il reste implicite (*cf.* FIG. 2.8) : une structure de contrôle est représentée au même titre qu'un calcul, et le modèle ne différencie pas *explicitement* traitements et routages. Cela permet donc dans une certaine mesure de représenter du routage périodique ; la limite vient du fait que, ne pouvant faire facilement

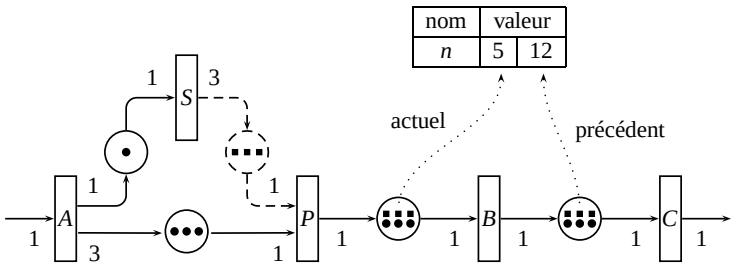


FIG. 2.9: Exemple de graphe SPDF. Les flots de données sont en traits pleins ; les jetons sont les disques noirs ; le flot de contrôle est en pointillés, ses jetons sont les carrés noirs.

cette distinction, il est difficile d'effectuer des transformations sur la topologie du graphe tout en préservant son comportement.

2.3.2 Graphes flots de données englobées synchrones

Le modèle flot de données englobées synchrone (*Synchronous Piggybacked Dataflow – SPDF*) [183, 184, 185] est une autre variante des graphes SDF. Il dissocie les canaux de données et de contrôle. Les nœuds de calcul peuvent avoir des états et paramètres internes, qui ne sont pas accessibles de l'extérieur. La mise à jour de ces paramètres se fait par le biais d'une table d'états globale : sur requête, un nœud met à jour ses paramètres par ceux présents dans la table. Cette méthode permet, entre autre, d'éviter des copies redondantes de jetons, ainsi que de diminuer les dimensions des mémoires : les jetons ont une granularité plus fine.

Détaillons ce principe sur l'exemple de la FIG. 2.9. Le producteur *A* produit trois données pour le nœud de *piggybacking P*, qui seront traitées par le nœud *B*. Ces traitements requièrent un paramètre *n* : si l'on modélise cette application en SDF, il faudra produire autant de jetons contenant *n* que de jetons données. Au lieu de quoi, en SPDF, *A* ne produit qu'un jeton paramètre pour le nœud de conversion d'état *S* ; *S* émet alors une requête pour ajouter la nouvelle valeur de *n* à la table d'états globale, et produit trois jetons de contrôle pour *P*. Ces jetons de contrôle représentent les pointeurs vers la nouvelle instance dans la table. Enfin, *P* associe à chaque jeton «donnée» le pointeur correspondant. Lorsque le couple donnée/pointeur est consommé par *B*, celui-ci met à jour sa valeur locale de *n*.

2.4 Flots de données à contrôle dynamique

Jusqu'à présent, nous avons étudié des modèles ne permettant de représenter que des traitements réguliers sur les données : soit des modèles dépourvus de structures de contrôle, soit des modèles à routage périodique. Nous envisageons maintenant des modèles plus complexes, où les productions et consommations des nœuds de calcul ne sont pas connues à la compilation. Les modèles de cette famille de tels modèles permettent d'exprimer un contrôle dynamique, à la façon de réseaux de Kahn [126]. Leur expressivité en est considérablement élargie, mais leur caractère Turing-complet les rend beaucoup plus difficiles à étudier statiquement [95, 145].

2.4.1 Flots de données à contrôle booléen

Les graphes flots de données à contrôle booléen (*Boolean-controlled Dataflow – BDF*), étudiés par Buck [45], ajoutent aux graphes SDF deux nœuds spécifiques de routage, en plus des nœuds de

calcul habituels. Nous les appellons *fusion* et *sélection*[4], et se comportent respectivement comme un multiplexeur et un démultiplexeur, séparant et rassemblant les flots de données.

Définition 2.28 (Graphes flots de données à contrôle booléen)**.** *Un graphe* flots de données à contrôle booléen *est une structure* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, C \rangle$ *où :*

- $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M \rangle$ *est un graphe marqué ;*
- $\mathcal{N}_s \subseteq \mathcal{N}$ *est un ensemble fini de nœuds de* sélection*, tel que* $\forall n \in \mathcal{N}_s : |n\bullet| = 2$ *et* $|\bullet n| = 1$ *;*
- $\mathcal{N}_f \subseteq \mathcal{N}$ *est un ensemble fini de nœuds de* fusion*, tel que* $\forall n \in \mathcal{N}_f : |n\bullet| = 1$ *et* $|\bullet n| = 2$ *;*
- *C est une fonction paramétrique, pouvant dépendre des données, qui détermine le nombre de jetons consommés sur les entrées des fusions ou produits sur les sorties des sélections, lors des exécutions de ces nœuds.*

Les nœuds *fusion* (resp. *sélection*) ont exactement deux places distinctes en entrée (resp. sortie), et exactement une place en sortie (resp. entrée). Le choix de la place où doit être consommé (resp. produit) un jeton est prise en interne, de façon similaire à une structure conditionnelle *si-alors-sinon* ; les branches *alors* et *sinon* sont respectivement étiquetées «1» et «0». Mais les prédicats ne sont pas explicites et peuvent éventuellement dépendre des données. Cela implique la perte de la propriété de décidabilité des équations de balance, du fait qu'il devient impossible de prédire dans quelles places seront lues ou produites des données, comme démontré par Buck [45]. Cette propriété reste décidable dans des cas particuliers [92], mais plus dans le cas général. En effet, un graphe BDF a l'expressivité d'une machine de Turing : il peut potentiellement s'exécuter à l'infini[5] et nécessiter une capacité mémoire également infinie. Cela implique que l'on ne peut dimensionner précisément et avec certitude les capacités des places.

Buck introduit également le modèle flot de données à contrôle entier (*Integer-controlled Dataflow – IDF*) [45, 46]. Il consiste en une extension de BDF incluant deux nouveaux types de nœuds : *Case* et *EndCase*. Ces derniers prennent en paramètre des valeurs entières, et permettent de modéliser des structures itératives de façon plus naturelle, ainsi que certains branchements conditionnels (semblables aux *switch-case* en langage C). Buck y donne un critère et un algorithme permettant de vérifier si le graphe est ordonnançable statiquement avec une mémoire bornée. Cependant, ce problème n'est en général pas décidable, là encore en raison du contrôle dépendant des données.

2.4.2 Fonctions basées sur flux

Dans sa thèse, Kienhuis développe le modèle de fonctions basées sur flux (*Stream Based Functions – SBF*) [131].

Définition 2.29 (Application basée sur flux)**.** *Une* application basée sur flux *est un graphe orienté* $G(V,E)$ *où :*

- *V est un ensemble d'objets exécutant des fonctions sur les flux (ou flots) de données.*
- *E est un ensemble de files par lesquelles transitent ces flux.*

Définition 2.30 (Objet basé sur flux)**.** *Un* objet basé sur flux *est un septulé* $\langle I, O, S, s, P, \omega, \mu \rangle$ *tel que :*

- *I est l'ensemble des* entrées *de l'objet,*
- *O est l'ensemble des* sorties *de l'objet,*

[4] Ces nœuds sont définis par l'auteur sous les noms respectifs de *Select* et *Switch*. Cette traduction libre est volontaire, de façon à conserver une cohérence avec des définitions ultérieures.

[5] *cf.* problème de l'arrêt.

- *S est l'*espace d'états $C \times D$*, avec C l'*espace de contrôle *et D l'*espace de données. *C représente l'ensemble des états de l'automate de contrôle, tandis que D représente l'ensemble des valeurs des variables locales,*
- *s est l'état courant, tel que* $s = (c,d) \in S$*,*
- *P est l'ensemble* $\{f_{init}, f_1, f_2, ..., f_x\}$ *des fonctions sur flux de données, de la forme :*

$$f : \begin{cases} I^m \times D \rightarrow O^n \times D \\ (i_0, ..., i_m, d) \mapsto (o_0, ..., o_n, d') \end{cases}$$

  f_{init} *est une fonction appelée uniquement à la création de l'objet, et permettant d'initialiser s,*
- ω *est la* fonction de transition*, déterminant l'état* c' *atteint après une exécution :*

$$\omega : \begin{cases} S \rightarrow C \\ s \mapsto c' \end{cases}$$

- μ *est la* fonction de fixation*, associant à chaque état de contrôle une et une seule fonction à exécuter :*

$$\mu : \begin{cases} C \rightarrow P \\ c \mapsto f_c \end{cases}$$

Par rapport au modèle AST de Backus [15], P est l'ensemble des fonctions, s est l'état, et (ω, μ) représente le contrôleur. Les lectures sur les entrées sont bloquantes et les écritures sur les sorties sont non-bloquantes. L'application peut être dynamique ; les conditions sur les données sont résolues à l'exécution. De ce fait, SBF pose les mêmes problèmes que BDF. Dans le même esprit de formalisation des réseaux de Kahn, citons encore les réseaux de processus dynamiques (*Reactive Process Networks*) [96], EIDF et CFDF [191].

2.4.3 Réseaux de processus flots de données

Les réseaux de processus flots de données (*Dataflow Process Networks – DPN*) [186] sont un cas particulier de DPN, inspiré des réseaux de Kahn : les règles sont ordonnées à la façon d'un *switch-case* d'un programme séquentiel : le nœud évalue les préconditions de ses règles de la première à la dernière, la première satisfaite étant utilisée pour le tirage. Notons que les lectures sur les entrées sont bloquantes, comme dans un réseau de Kahn.

2.5 Flots de données à contrôle paramétré

Les modèles à contrôle paramétré [203, 204, 231] sont des compromis entre modèles à contrôle statique et modèles Turing-complets. Leur expressivité est plus large que les premiers : les motifs des listes de poids ou conditions de routage dépendent de paramètres entiers, *a priori* inconnus à la compilation. Cependant, leurs équations de balance peuvent être résolues par calcul symbolique. Ils conservent ainsi le caractère décidable de certaines propriétés, telles que l'absence de blocage, la vivacité et le caractère borné.

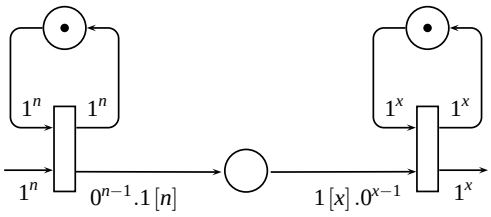


FIG. 2.10: Exemple d'encodeur/décodeur en CDDF. Les variables entre crochets correspondent aux valeurs attribuées aux jetons de contrôle.

2.5.1 Flots de données évolutifs

L'exemple le plus simple d'un tel contrôle paramétré est celui des flots de données évolutifs (*Scalable Synchronous Dataflow – SSDF*) [203], conçus en tant que modèle intermédiaire pour la compilation sur DSP. De même qu'en SDF, un poids est associé à chaque entrée et sortie d'un nœud SSDF. Lors d'une exécution, le nombre de jetons effectivement produit ou consommé par un nœud est un multiple de ce poids, selon un paramètre défini par l'ordonnanceur. Cette approche permet de diminuer le nombre d'activation de chaque nœud, et ainsi de diminuer de façon optimale le surcoût introduit par les changements de contexte du processeur [204].

2.5.2 Flot de données cyclo-dynamique

Par la suite, Wauters *et al.* [231] présentent le modèle flot de données cyclo-dynamique (*Cyclo-Dynamic Dataflow – CDDF*), version paramétrée de CSDF. Tout nœud a un comportement périodique ; une période est une séquence de *phases*. Les productions et consommations de jetons au cours d'une phase, et donc les tirages des nœuds, dépendent de l'indice dans la séquence et de la valeur de jetons dits *de contrôle* ; un nœud n'est pas supposé se comporter tel un automate, dont des variables internes influeraient sur l'exécution. Cela signifie, par exemple, que si un nœud consomme n jetons sur l'une de ses entrées à un instant donné, alors un jeton de contrôle contenant la valeur n doit être consommé sur une autre entrée au sein de ce même instant ; la valeur n'est pas stockée d'un instant à l'autre. En revanche, la valeur d'un jeton de contrôle peut déterminer la longueur d'une phase. Les auteurs y expliquent comment la consistance (mémoires bornées et comportement périodique) du graphe peut être prouvée hors-ligne, indépendamment des valeurs des jetons de contrôle, par calcul symbolique des équations de balance. Cependant, il demeure impossible de calculer les tailles maximales des mémoires sans plus de précisions sur ces valeurs.

Exemple 2.31. Pour illustrer ce point, nous reprenons en FIG. 2.10 l'un des exemples avancés par Wauters *et al.* [231]. Les symboles entre crochets correspondent aux valeurs des jetons de contrôle. L'idée est la suivante : le nœud de gauche lit n données, et les encode en un jeton qu'il transmet lors de sa $n^{\text{ème}}$ exécution. Le décodeur, à droite, reçoit le jeton de contrôle. Le nombre de jetons produits par le décodeur dépend de la valeur x portée par ce jeton de contrôle. En l'occurrence, nous obtenons l'identité $x = n$.

2.6 Hiérarchie et métamodèles

La composition hétérogène et hiérarchique des modèles précédents est abordée par divers travaux, notamment sur *charts[6] [98] et DFCharts [199]. Ces deux modèles mélangent graphes SDF et automates, offrant ainsi différents niveaux de raffinement du modèle et de ses composants. Le premier a une approche par couches homogènes : les graphes SDF sont composés entre eux uniquement, de même pour les automates. Au contraire, le second étudie la composition de différents modèles au sein d'une même couche.

L'approche par métamodèles offre une modélisation hiérarchique pouvant être appliquée à tous les modèles précédemment cités. Elle permet une reconfiguration structurée des flots de données : le système est contrôlé par un ensemble de paramètres. Ces paramètres peuvent changer à l'exécution pour chaque période du graphe ; cependant, une fois fixés, ils ne peuvent être modifiés avant la fin de la période courante. Ainsi, le comportement du graphe peut changer dynamiquement. Son ordonnancement est quasi-statique, dépendant de ces paramètres : il est défini à la compilation, mais les nombres exacts de tirages des nœuds sont calculés à l'exécution. Des exemples de tels métamodèles sont les flots de données paramétrés [27, 28, 29] et assimilés (BLDF [136], DGT [136, 137]), ou encore HPDF [115, 211, 212].

[6] Prononcé «*starcharts*».

Chapitre 3

Graphes à routage *k*-périodique

Tous les modèles sont faux, mais certains sont utiles.

George E. P. Box et Norman R. Draper, *Empirical Model-Building and Response Surfaces*, 1987.

Sommaire

Contrairement aux graphes CSDF, les graphes à routage *k*-périodique (*k-periodically routed graph – KRG*) [33, 37] furent introduits dans l'optique de rendre explicite le contrôle, et le distinguer des nœuds de calcul. Les KRG aiguillent leurs jetons par le biais de deux types de nœuds additionnels, *sélection* et *fusion*, annotés par des séquences binaires ultimement périodiques. Elles déterminent dans quelles places, en entrée et en sortie, ces nœuds doivent consommer ou produire les jetons.

L'intérêt de ce modèle est multiple. Tout d'abord, de la même façon qu'un graphe marqué permet de modéliser un système insensible aux latences (*cf.* sous-section 2.2.3), un KRG peut modéliser un tel système enrichi par du routage de données. Ensuite, afin de s'assurer que le système est synthétisable, sa sémantique est assez riche pour permettre de vérifier des propriétés fondamentales, telles que le caractère borné, l'absence de blocages, ou encore le calcul d'un ordonnancement, et ce de façon statique. Enfin, nous définissons un ensemble de transformations de sa topologie qui préservent ces propriétés et la fonctionnalité du graphe.

3.1 Présentation

Dans cette section, nous donnons tout d'abord les définitions des KRG et de la sémantique de leurs nœuds, avant de préciser leurs règles d'activation et de tirage. Nous expliquons comment vérifier le caractère borné d'un KRG par abstraction en graphe SDF. Enfin, nous présentons une première solution pour vérifier la vivacité d'un KRG.

3.1.1 Définition

Définition 3.1 (Graphe à routage *k*-périodique)**.** *Un graphe à routage k-périodique est un graphe orienté biparti* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$*, où :*

- $\mathcal{N}$ *est un ensemble fini de nœuds, répartis en quatre sous ensembles distincts :*
  - $\mathcal{N}_\lambda$ *est l'ensemble des nœuds de* calcul.
  - $\mathcal{N}_c$ *est l'ensemble des nœuds de* copie*, tels que :*

$$\forall n \in \mathcal{N}_c, \ \ |\bullet n| = 1 \ \ et \ \ |p\bullet| > 1 \tag{3.1}$$

  - $\mathcal{N}_s$ *est l'ensemble des nœuds de* sélection*, tels que :*

$$\forall n \in \mathcal{N}_s, \ \ |\bullet n| = 1 \ \ et \ \ |p\bullet| = 2 \tag{3.2}$$

  - $\mathcal{N}_f$ *est l'ensemble des nœuds de* fusion*, tels que :*

$$\forall n \in \mathcal{N}_f, \ \ |\bullet n| = 2 \ \ et \ \ |p\bullet| = 1 \tag{3.3}$$

- $\mathcal{P}$ *est un ensemble fini de* places*, telles que :*

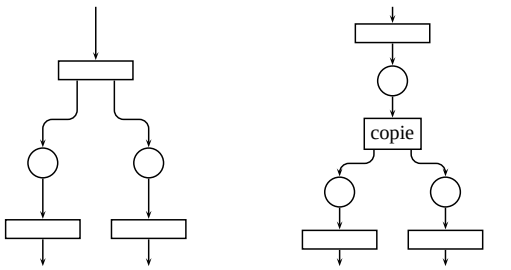
$$\forall p \in \mathcal{P}, \ \ |\bullet p| = |p\bullet| = 1 \tag{3.4}$$

- $\mathcal{T}$ *est un ensemble fini d'arcs reliant nœuds et places :* $\mathcal{T} \subseteq (\mathcal{N} \times \mathcal{P}) \cup (\mathcal{P} \times \mathcal{N})$
- *M est la fonction attribuant un marquage initial à chaque place.*

$$M : \mathcal{P} \to \mathbb{N} \tag{3.5}$$

- *R est la fonction attribuant une séquence de routage aux nœuds de sélection et fusion.*

$$R : \mathcal{N}_s \cup \mathcal{N}_f \to \mathbb{P} \tag{3.6}$$



(a) Production de deux données distinctes.

(b) Une seule donnée est produite, puis dupliquée.

FIG. 3.1: Rôle du nœud de copie.

D'un point de vue structurel, $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M \rangle$ est un graphe marqué. Cependant ,les nœuds d'un KRG ont deux vocations distinctes. Les nœuds de calculs, d'une part, effectuent des traitements sur les données, et sont dépourvus de contrôle. En cela, ils sont semblables aux nœuds des graphes marqués. D'autre part, les nœuds de sélection, de fusion et de copie, sont quant à eux des nœuds de transport : ils permettent d'enrichir la modélisation du contrôle en explicitant les routages et duplications des données. Ainsi, nous pouvons étudier le transport des jetons indépendamment de leurs traitements. En cela, un KRG offre un raffinement d'un graphe CSDF. Détaillons maintenant chaque élément de la définition :

Nœuds de copie. La nature (ou multiplicité) des jetons doit être prise en compte. Bien que les données sous-jacentes soient abstraites, il existe une différence sémantique importante illustrée par la FIG. 3.1. De la même manière que les KRG introduisent un routage explicite par rapport à CSDF par des nœuds dédiés, nous souhaitons être à même de faire la distinction entre la duplication d'une même donnée et la production de deux données distinctes. Dans le cas 3.1(a), le producteur a deux sorties distinctes, chacune vers un unique consommateur : le nœud de calcul produit deux données potentiellement différentes. Dans le cas 3.1(b), le producteur n'a qu'une seule sortie : une seule donnée est produite puis dupliquée, pour chaque consommateur. Une telle information est nécessaire pour effectuer des transformations sur la topologie du graphe (*e.g.* factorisation ou expansions locales) ou le routage de ses données.

Routage. Les nœuds de sélection et de fusion sont assimilables, respectivement, à des démultiplexeurs 1 vers 2 et des multiplexeur 2 vers 1. Dans le principe, ils sont semblables aux nœuds de routage de BDF. Cependant, à chaque nœud de sélection ou de fusion est associée une séquence binaire ultimement périodique, par la fonction R. Le routage des jetons au cours de l'exécution du graphe est donc imposé par un motif prédéterminé. La lettre 0 (resp. 1) de la séquence correspond à un aiguillage *depuis* ou *vers* la place étiquetée 0 (resp. 1).

Places et jetons. Les places et les arcs qui les relient aux nœuds représentent des connexions point-à-point entre blocs de traitement. Les places d'un KRG se comportent comme des files : l'ensemble des jetons traversant chacune des places est totalement ordonné. Par ailleurs, le marquage initial représente les données initiales du système. Étant donné qu'une fonction n'est pas forcément inversible, nous considérons donc qu'il est imposé à la conception et qu'il est constant[1].

Les places ont un unique producteur et un unique consommateur, et les jetons sont consommés de

[1]Dans une moindre mesure, certains jetons initiaux peuvent être propagés statiquement dans le système à la façon d'une propagation ou duplication de constantes, mais ils ne peuvent en aucun cas être rétropropagés

façon déterministe et sans conflit (*cf.* sous-section 2.1). Ainsi, l'exécution de n'importe quel nœud ne désactive aucun autre nœud que lui-même. Les principes d'exécution du graphe sont définis comme suit :

Règle 3.2 (Activation)**.**
- *Un nœud de calcul, de copie ou de sélection est activé si et seulement si chaque place en entrée contient au moins un jeton.*
- *Un nœud de fusion est activé si et seulement si sa place en entrée, indiquée par la première lettre de la séquence de routage, contient au moins un jeton.*

Règle 3.3 (Tirage)**.**
- *Seul un nœud activé peut être tiré.*
- *Le tirage d'un nœud calcul ou de copie consomme un jeton dans chaque place en entrée. Un jeton est produit dans chaque place en sortie.*
- *Le tirage d'un nœud de fusion consomme un jeton dans la place indiquée par la séquence de routage. Un jeton est produit dans la place en sortie. La première lettre de la séquence de routage est consommée.*
- *Le tirage d'un nœud de sélection consomme un jeton dans sa place en entrée. Un jeton est produit dans la place en sortie, indiquée par la première lettre de la séquence de routage. Celle-ci est consommée.*

Notons que la lecture d'un nœud de fusion est *bloquante* : les places dans lesquelles sont consommés les jetons dépendent uniquement de la séquence de routage, et non d'un test sur la présence de jeton dans une place. Ainsi, le tirage d'un nœud de fusion ne peut dépendre de l'ordre d'arrivée des jetons sur ses entrées.

3.1.2 Sémantique et flots de jetons

Afin d'éviter des interprétations erronées de la définition d'un KRG et de ses règles d'activation et de tirage, nous définissons formellement les relations entre jetons. Pour cela, nous introduisons tout d'abord la notion de *flot de jetons*, avant d'en étudier le comportement.

Définition 3.4 (Flot de jetons)**.** *Soit $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ un KRG, et $t = (x,y)$ un arc de $\mathcal{T}$. On étiquette par l'indice t_i, avec $i \in \mathbb{N}^*$, le $i^{ème}$ jeton traversant t. Ces indices définissent une suite d'entiers, appelée le* flot de jetons *traversant t.*

Nous devons maintenant exprimer les relations entre jetons produits et consommés. La sémantique des KRG et les règles d'activation et de tirage induisent des relations entre flots de jetons. Ces différentes relations dépendent de la nature des nœuds traversés. En effet, les flots subissent des traitements qui dépendent de leurs sémantiques et de leurs règles de tirage (calcul ou routage). Par exemple, pour une place p, son $i^{\text{ème}}$ jeton entrant est aussi son $(i + M(p))^{\text{ème}}$ jeton sortant, compte tenu du marquage initial. L'ensemble de ces primitives est donné par la définition 3.5.

Définition 3.5 (Relation élémentaire entre flots)**.** *Soit $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ un KRG, et $t = (x,y)$ et $t' = (y,z)$ deux arcs de $\mathcal{T}$. Considérons un jeton produit par x et routé vers z, via y, sur le chemin $t \rightsquigarrow t'$. Il existe un ensemble de relations de flot de la forme $\curvearrowright_y \subset \mathbb{N}^* \times \mathbb{N}^*$, telle que $i \curvearrowright_y j$, où i est l'indice du jeton*

dans le flot de t, et j est son indice dans le flot de t' :

$$\forall p \in \mathcal{P},\ \forall i \in \mathbb{N}^*, \qquad i \curvearrowright_p i + M(p) \tag{3.7}$$

$$\forall n \in \mathcal{N}_\lambda \cup \mathcal{N}_c,\ \forall i \in \mathbb{N}^*, \qquad i \curvearrowright_n i \tag{3.8}$$

$$\forall s \in \mathcal{N}_s,\ \forall b \in \mathbb{B},\ \forall i \in \mathbb{N}^*, \qquad \mathrm{idx}_b(R(s), i) \curvearrowright_s^{(b)} i \tag{3.9}$$

$$\Leftrightarrow \qquad i \curvearrowright_s^{(b)} |\mathrm{prf}(R(s), i)|_b \ \wedge\ (R(s))_i = b \tag{3.10}$$

$$\forall f \in \mathcal{N}_f,\ \forall b \in \mathbb{B},\ \forall i \in \mathbb{N}^*, \qquad i \curvearrowright_f^{(b)} \mathrm{idx}_b(R(f), i) \tag{3.11}$$

$$\Leftrightarrow \qquad |\mathrm{prf}(R(f), i)|_b \curvearrowright_f^{(b)} i \ \wedge\ (R(f))_i = b \tag{3.12}$$

où b fait référence à l'entrée ou à la sortie considérée du nœud de routage.

Ces relations sont monotones : $\forall (i_1, j_1), (i_2, j_2) \in \curvearrowright,\ i_1 \leq i_2 \Leftrightarrow j_1 \leq j_2$. Les places, nœuds de copie et de calcul n'altèrent pas l'ordre des jetons ; les relations qui leurs sont associées sont des bijections entre l'indice d'un jeton en entrée et celui en sortie. Pour les nœuds de sélection et de fusion, les relations dépendent des séquences de routage, selon que les jetons empruntent l'un ou l'autre chemin. Pour une sélection, les relations sont surjectives ; elles sont injectives pour une fusion.

Maintenant que nous sommes en mesure de calculer la relation entre les jetons sortant d'une place et ceux entrant dans une suivante immédiate, nous pouvons composer ces relations élémentaires afin de calculer la relation sur l'ensemble d'un chemin donné du KRG :

Définition 3.6 (Relation entre flots sur un chemin)**.** *Soit Σ l'ensemble des chemins d'un KRG. Un chemin $\sigma \in \Sigma$ est de la forme $\sigma = p_1.n_1.p_2.\ldots.n_l.p_{l+1}$, avec $n_1, \ldots, n_l \in \mathcal{N}$ et $p_1, \ldots, p_{l+1} \in \mathcal{P}$. La relation $\curvearrowright_\sigma$ sur le chemin σ entre p_1 et p_{l+1} peut être calculée par composition des relations élémentaires :*

$$\curvearrowright_\sigma = \curvearrowright_{p_{l+1}} \circ \curvearrowright_{n_l} \circ \ldots \circ \curvearrowright_{p_2} \circ \curvearrowright_{n_1} \tag{3.13}$$

où $\curvearrowright_b \circ \curvearrowright_a = \{(i,k) \mid \exists (i,j) \in \curvearrowright_a, (j,k) \in \curvearrowright_b\}$.

3.1.3 Caractère borné

Comme nous l'avons vu en sous-section 2.1, vérifier le caractère borné d'un KRG est de prime importance pour sa synthèse en circuit : le problème consiste à trouver un équilibre entre les productions et consommations de jetons dans chaque place. Il peut se résumer à : «existe-t-il une séquence de tirages telle que le nombre de jetons dans chaque place reste borné ?»

La réponse dépend à la fois du rythme de tirage de chaque nœud et du routage. Considérons le cas particulier d'un KRG où tous les nœuds sont des calculs ou des copies : ce KRG est équivalent à un graphe marqué. Le théorème 2.7 nous permet donc de conclure que dans une séquence de tirages dont la longueur tend vers l'infini, le rapport entre les nombres de tirages de chaque couple de nœuds tend vers 1.

Dans le cas général, un KRG contient des nœuds de sélection et de fusion, dont les consommations et productions de jetons varient périodiquement. Déterminer les rythmes de tirage de chacun demande alors à intégrer leurs productions et consommations sur une période de leur séquence de routage. Cette intégration est réalisée au moyen d'une abstraction du KRG en graphe SDF (*cf.* section 3.1.4). Puis, le rythme de chaque nœud est calculé par résolution des équations de balance sur l'abstraction en SDF. Une solution du système (2.21) est un ensemble de coefficients x_n, pour tout nœud n, tel que :

- Si n est un nœud de calcul ou de copie, alors il est exécuté x_n fois au cours d'une période ;
- Si n est un nœud de sélection ou de fusion, alors il est exécuté $x_n \times |R(n)|$ fois au cours d'une période.

3.1.4 Abstraction du KRG en graphe SDF

Comme mentionné ci-dessus, l'abstraction d'un KRG en SDF repose essentiellement sur une intégration du nombre de jetons routés *depuis* ou *vers* une place, au cours d'une période d'un nœud de routage.

Règle 3.7 (Abstraction d'un KRG en SDF)**.** *On construit l'abstraction d'un KRG* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ *en un graphe SDF* $\langle \mathcal{N}', \mathcal{P}', \mathcal{T}', M', W \rangle$ *par :*

$$\mathcal{N}' \stackrel{\text{def}}{=} \mathcal{N} \tag{3.14}$$
$$\mathcal{P}' \stackrel{\text{def}}{=} \mathcal{P} \tag{3.15}$$
$$\mathcal{T}' \stackrel{\text{def}}{=} \mathcal{T} \tag{3.16}$$
$$M' \stackrel{\text{def}}{=} M \tag{3.17}$$

Pour toute place p *de* $\mathcal{P}$*, on définit* $W(p) \stackrel{\text{def}}{=} (w_o, w_d)$ *tels que :*

$$w_o \stackrel{\text{def}}{=} \begin{cases} 1 & si \bullet p \in \mathcal{N}_\lambda \cup \mathcal{N}_c \\ |R(\bullet p)|_b & si \bullet p \in \mathcal{N}_s \\ |R(\bullet p)| & si \bullet p \in \mathcal{N}_f \end{cases} \qquad w_d \stackrel{\text{def}}{=} \begin{cases} 1 & si\ p\bullet \in \mathcal{N}_\lambda \cup \mathcal{N}_c \\ |R(p\bullet)| & si\ p\bullet \in \mathcal{N}_s \\ |R(p\bullet)|_b & si\ p\bullet \in \mathcal{N}_f \end{cases} \tag{3.18}$$

où $b \in \mathbb{B}$ *représente l'étiquette de la sortie (resp. de l'entrée) du nœud de sélection (resp. de fusion).*

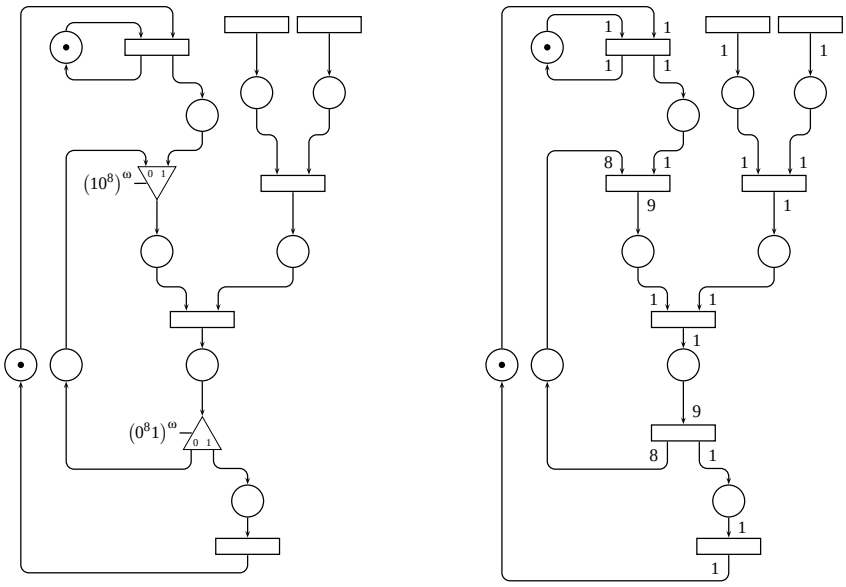
La topologie du graphe reste identique ; les nœuds de sélection (resp. de fusion) sont abstraits par des nœuds de calcul ayant exactement une entrée et deux sorties (resp. deux entrées et une sortie). Sur une période, un nœud de sélection n consomme $|R(n)|$ jetons et en produit respectivement $|R(n)|_0$ et $|R(n)|_1$ en sortie. De la même manière, un nœud de fusion n' est abstrait par un nœud consommant respectivement $\left|R\left(n'\right)\right|_0$ et $\left|R\left(n'\right)\right|_1$ jetons et en produisant $\left|R\left(n'\right)\right|$. Les nœuds de calcul restent inchangés, consommant un jeton sur chacune de leurs entrées et en produisant un sur chacune de leurs sorties lors d'une exécution. Un exemple est donné en FIG. 3.2.

Proposition 3.8. *Un KRG est borné si et seulement si son abstraction en SDF l'est.*

3.1.5 Vivacité

Des blocages peuvent potentiellement se produire lorsque certaines places ne contiennent pas de jetons. Par exemple, en FIG. 3.2(a), concaténer le préfixe «0» à la séquence de routage de la fusion revient à introduire une situation de *verrou mortel* : le jeton est orienté vers le chemin de droite, alors que la fusion l'attend du côté gauche. La détection de telles erreurs est d'une importance primordiale. Nous ne cherchons a y répondre que dans le cas de KRG bornés. Contrairement à [33], nous ne parlerons pas de vivacité, qui est une propriété trop forte pour la plupart des applications usuelles, mais de quasi-vivacité. Par exemple, il est courant qu'une portion de code ne soit exécutée qu'à l'initialisation ou la terminaison du système. Ce problème est par conséquent plus fonctionnel que structurel.

Une première solution, la plus intuitive, consiste à réaliser une simulation de l'exécution du graphe jusqu'à la fin de la première période [33]. Si le KRG est borné et si ses routages sont périodiques, alors son espace des états atteignables est fini. Ainsi, la simulation se termine toujours. On exécute donc le graphe jusqu'à aboutir à l'une des situations suivantes :



(a) (b)

FIG. 3.2: Abstraction d'un (a) KRG en (b) graphe SDF.

– La simulation se termine du fait que plus aucun nœud n'est activé. Il y a blocage : le graphe souffre certainement d'un défaut de conception (topologie, séquences de routage, marquage initial) ;
– À un certain pas d'exécution, la simulation boucle et retourne dans un état du graphe déjà exploré. Le graphe est quasi-vivant.

Nous voyons dans la section suivante comment répondre à cette question en termes formels, par le biais d'une analyse statique des dépendances sur les flots de jetons.

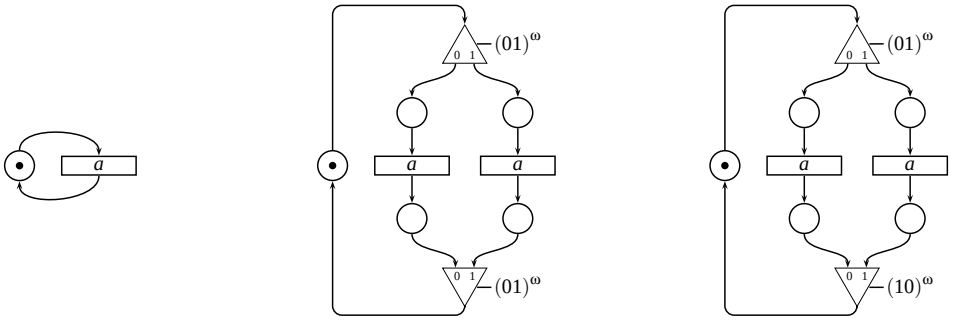
3.2 Dépendances de données et forme normale

Dans cette section, nous définissons une première forme normale de KRG. Sa construction demande à étudier les dépendances entre jetons parcourant un KRG. Dans ce but, nous définissons deux représentations intermédiaires, des graphes de dépendance. Le premier, ou graphe de dépendance étendu, est potentiellement *infini*. Nous étudions alors les propriétés nécessaires à la construction d'une forme *finie* de graphe de dépendance, le graphe de dépendance réduit. C'est à partir de ce second modèle que nous construisons la forme normale.

3.2.1 Motivation

Nous partons de deux constats. Tout d'abord, deux KRGs peuvent avoir des topologies différentes, et pourtant avoir la même fonctionnalité, c'est-à-dire appliquer les mêmes transformations à leurs jetons. À l'inverse, deux graphes peuvent avoir des topologies identiques, mais des comportements différents à cause de leurs routages.

Exemple 3.9. Prenons pour exemple les KRG de la FIG. 3.3. Les KRG (a) et (b) n'ont certes pas les mêmes topologies, mais il est évident que tous les jetons subissent, et dans le même ordre, une



(a) Calculs séquentiels. (b) Calculs parallèles. (c) Calculs parallèles avec verrou.

FIG. 3.3: Exemples de KRG. (a) et (b) ont des topologies différentes mais appliquent les mêmes opérations à leurs jetons. (b) et (c) ont des topologies identiques, mais (c) n'est pas vivant.

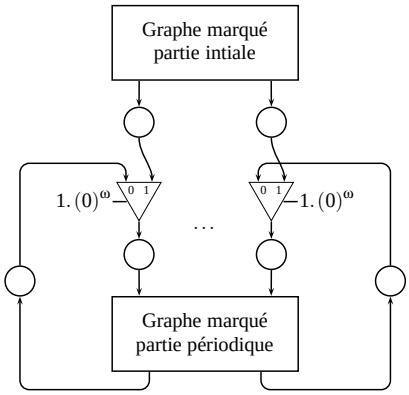


FIG. 3.4: Aspect d'un quasi-graphe marqué.

transformation nommée *a*. Au contraire, (b) et (c) ont pourtant la même topologie, mais (b) est vivant, tandis que (c) souffre d'un verrou.

D'autre part, les routages d'un KRG évoluent au cours de son exécution, par la consommation progressive des séquences de routage. Ainsi, deux jetons transitant successivement par une place ne suivent pas nécessairement le même chemin. Le fait que tous les routages soient ultimement périodiques nous apporte cependant un élément supplémentaire : après une séquence de tirages suffisamment longue dans un KRG vivant, tout jeton empruntera nécessairement le même chemin que l'un de ses prédécesseurs. Intuitivement, nous pouvons penser que cela se produira une fois que chaque nœud de routage aura été tiré autant de fois que le *ppcm* (plus petit commun multiple) des longueurs des périodes des routages, à un décalage près (pour la phase initiale).

Nous voulons donc définir une forme normale de KRG, de taille finie, et une notion d'équivalence entre KRG : deux KRG sont équivalents si et seulement s'ils ont la même forme normale. Nous montrons par la suite que cette forme normale est un *quasi-graphe marqué* : la FIG. 3.4 illustre ce principe : deux graphes marqués, l'un cyclique, l'autre acyclique, sont reliés par une «couche» de nœuds de fusion ; les séquences de routage sont toutes de la forme $1.(0)^{\omega}$.

Définition 3.10 (Quasi-graphe marqué)**.** *Soit* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ *un KRG. On dit que ce graphe est un quasi-graphe marqué si et seulement s'il existe les ensembles* $\mathcal{N}' \subseteq \mathcal{N}_\lambda \cup \mathcal{N}_c$, $\mathcal{P}' \subseteq \mathcal{P}$, *et* $\mathcal{T}' \subseteq \mathcal{T}$ *tels que les conditions suivantes soient remplies :*

– $\mathcal{N}_s = \emptyset$,
– $\langle \mathcal{N}' \cup \mathcal{N}_f, \mathcal{P}', \mathcal{T}', M' \rangle$ *est un graphe marqué acyclique,*
– $\langle \mathcal{N} \setminus \mathcal{N}', \mathcal{P} \setminus \mathcal{P}', \mathcal{T} \setminus \mathcal{T}', M'' \rangle$ *est un graphe marqué,*
– $\forall n \in \mathcal{N}_f,\ {\bullet n} \not\subset \mathcal{P}' \ \wedge\ {\bullet n} \not\subset (\mathcal{P} \setminus \mathcal{P}') \ \wedge\ n{\bullet} \in (\mathcal{P} \setminus \mathcal{P}')$.

où M' et M'' sont respectivement les restrictions de M à $\mathcal{P}'$ et $\mathcal{P} \setminus \mathcal{P}'$.

En quelque sorte, la construction d'un quasi-graphe marqué revient à retirer tout le routage d'un KRG, notamment en «déroulant» ses boucles sur une hyperpériode ; seuls les nœuds de fusions faisant office de séparation entre parties initiale et périodique sont préservés (ou introduits). Pour ce faire, étudier la topologie du KRG est insuffisant : routage et marquage initial doivent être pris en compte. La construction d'une forme normale demande à étudier plus précisément les relations *entre les jetons* : autrement dit, les *dépendances de données*. Les sous-sections suivantes détaillent les différentes étapes de la construction de la forme normale.

3.2.2 Dépendances entre jetons

De la sémantique d'un KRG résulte un ensemble de dépendances entre flots de jetons, et même au sein d'un flot. Une dépendance de p_i vers q_j est notée $p_i \rightarrow q_j$. Ces dépendances symbolisent la causalité ou la séquentialité des opérations du KRG. Par exemple, un opérande doit lui-même être calculé avant de servir à un autre calcul ; ou un producteur doit d'abord émettre une première donnée avant une suivante. Une étude fine de ces dépendances permet, par la suite, de déterminer si un KRG est exempt de blocages ou non, ou si des transformations de sa topologie sont susceptibles d'en introduire.

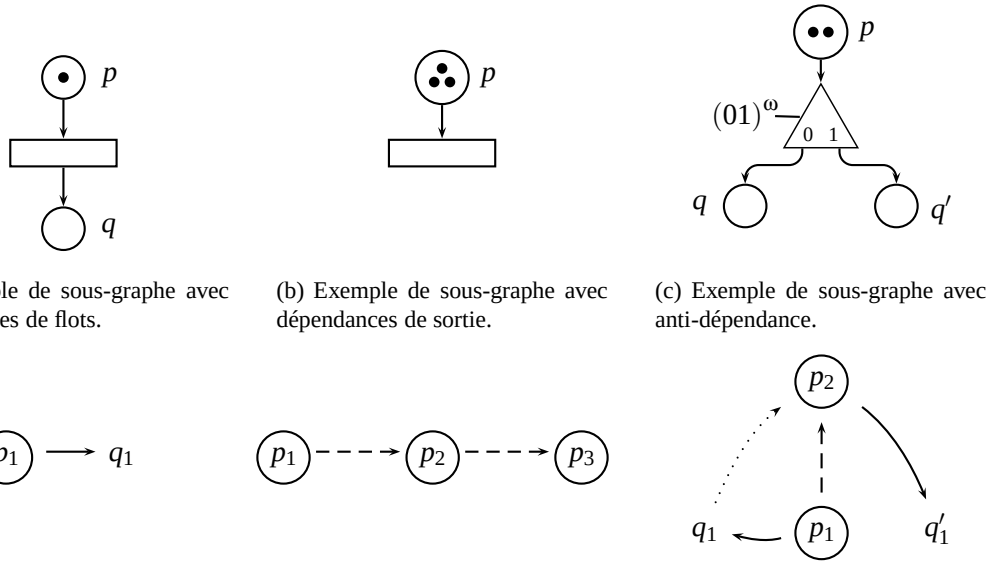
De nombreux travaux ont été réalisés sur l'analyse des dépendances de données, dans les domaines de compilation et de parallélisation automatique de programmes. Nous renvoyons le lecteur à Darte *et al.* [72] pour un état de l'art. Par ailleurs, ce genre d'outils d'analyse est utilisé pour des graphes SDF sous le nom de *graphes de précédence* [189, 214].

Certaines de ces techniques peuvent être transposées à notre problématique de dépendances de jetons. Pour un KRG donné, l'ensemble de ses dépendances peut-être représenté sous la forme d'un *graphe de dépendances*. En effet, graphes de dépendance et réseaux de Petri sans conflit (et dérivés) sont analogues et offrent la même expressivité [110, 116]. Plus précisément, ce sont des formes duales : un réseau de processus modélise des relations entre opérations, tandis que le graphe de dépendances représente les relations entre données. Transposées dans le contexte d'un KRG, les dépendances de données sont les suivantes :

Dépendance de flot. *(consommation → production)* En compilation, une dépendance de flot entre deux instructions A et B correspond à l'écriture d'une donnée dans un registre par A, lue ultérieurement par B. Pour un KRG, nous dirons qu'il existe une telle dépendance si pour un nœud n quelconque et deux places $p \in {\bullet n}$ et $p' \in n{\bullet}$, la consommation du jeton p_i entraîne la production du jeton p'_j. Ou plus simplement, il existe une dépendance de flot de p_i vers p'_j, notée $p_i \rightarrow_f q_j$[2] si et seulement si $i \curvearrowright_n j$. Le graphe de la FIG. 3.5(a) donne un exemple de dépendance de flot. Le jeton p_1 est un marquage initial. Il est consommé par le nœud pour produire q_1. Les dépendances sont représentées en FIG. 3.5(d).

Dépendance de sortie. *(production → production)* Une dépendance de sortie correspond à deux écritures consécutives dans une ressource partagée. En l'occurrence, les places se comportant

[2] On peut préciser que la dépendance est causée par le tirage du nœud n en notant $p_i \xrightarrow{n}_f q_j$, ou directement sa fonctionnalité par $p_i \xrightarrow{\lambda(n)}_f q_j$.



(a) Exemple de sous-graphe avec dépendances de flots.

(b) Exemple de sous-graphe avec dépendances de sortie.

(c) Exemple de sous-graphe avec anti-dépendance.

(d) Dépendances de flots entre les jetons de p et q, en trait plein, pour le cas (a).

(e) Dépendances de sortie entre jetons de p, en traits pointillés, pour le cas (b).

(f) Anti-dépendance entre q_1 et p_2, pour le cas (c).

FIG. 3.5: Exemples des différentes dépendances entre jetons. Les jetons entourés sont initiaux.

comme des FIFO, la tête d'une FIFO peut être vue comme ladite ressource partagée. Ainsi, pour toute place p, il existe une dépendance de sortie entre les jetons p_i et p_{i+1}, notée $p_i \rightarrow_s q_j$: le $i^{\text{ème}}$ jeton doit sortir de p avant que n'en sorte le $(i+1)^{\text{ème}}$. Le graphe de la FIG. 3.5(b) en donne un exemple. Un jeton ne peut sortir d'une place que si l'ensemble de ses prédécesseurs en sont eux-mêmes sortis ; le passage de p_i permet le passage de p_{i+1}, et ainsi de suite. Les dépendances sont représentées en FIG. 3.5(e).

Anti-dépendance. *(production → consommation)* La notion d'anti-dépendance, dans un KRG, résulte d'une combinaison de dépendances de flot et de sortie. En compilation, une anti-dépendance entre deux instructions A et B signifie que A lit une donnée dans une ressource partagée avant que B n'y écrive, et écrase ainsi la donnée. Dans le cas d'un KRG, nous ne pouvons perdre de données par écrasement : ce concept est interdit dans notre modèle. Cependant, comme nous l'avons mentionné au point précédent, un canal de communication peut être vu comme une ressource partagée par plusieurs jetons. Pour qu'un jeton «libère» la tête du canal, il doit être consommé : la production du nouveau jeton permet au second jeton du canal d'en prendre la tête. Cela signifie que si deux jetons p_i et p_{i+1} sont en séquence (dépendance de sortie $p_i \rightarrow_s p_{i+1}$), et si p_i est lui-même consommé pour le calcul de q_j (dépendance de flot $p_i \rightarrow_f q_j$), alors il existe une anti-dépendance de q_j vers p_{i+1} : la consommation de p_i pour le calcul de q_j permet la production de p_{i+1}. Le graphe de la FIG. 3.5(c) mène à une telle situation, représentée en FIG. 3.5(f).

Étant donné que les anti-dépendances ne sont, dans notre modèle, qu'une *conséquence* des dépendances de flot et de sortie, nous attacherons un intérêt particulier à l'étude de ces deux dernières ; les anti-dépendances correspondantes en seront déduites.

3.2.3 Graphe de dépendance étendu

Pour étudier les dépendances de données, nous devons les «dérouler». L'exécution d'un KRG quasi-vivant est infinie ; il en est de même pour le nombre de jetons transitant par certaines places. Nous définissons donc le concept de *graphe de dépendance étendu* (GDE). Ce graphe de dépendance est potentiellement infini, puisqu'il détaille l'ensemble des dépendances au cours d'une exécution de KRG.

Définition 3.11 (Graphe de dépendance étendu)**.** *Soit $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ un KRG. Son graphe de dépendance étendu est un quadruplet $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$ tel que :*

- *$\mathcal{M}$ est un ensemble (éventuellement infini) de sommets, correspondant aux jetons p_i transitant par chaque place p.*
- *$\mathcal{D}$ est un ensemble (éventuellement infini) d'arcs, ou* dépendances*, avec $\mathcal{D} \subseteq \mathcal{M} \times \mathcal{M}$. On note $\mathcal{D}_f$ le sous-ensemble des dépendances de flot, et $\mathcal{D}_s$ le sous-ensemble des dépendances de sortie, tels que $\forall\, p, q \in \mathcal{P},\ p\bullet = \bullet q$,*

$$\begin{aligned} \forall\, p_i, q_j \in \mathcal{M}, \qquad & i \curvearrowright_q \circ \curvearrowright_{p\bullet} j \ \Leftrightarrow\ p_i \overset{p\bullet}{\rightarrow}_f q_j \\ \forall\, p_i, p_{i+1} \in \mathcal{M}, \qquad & p_i \rightarrow_s p_{i+1} \end{aligned}$$

- *$\mathcal{I}$ est l'ensemble fini des jetons initiaux, avec $\mathcal{I} \subseteq \mathcal{M}$:*

$$\mathcal{I} = \{\, p_i \mid p_i \in \mathcal{M},\ i \leq M(p) \}$$

- *$\mathcal{Y}$ est l'ensemble (éventuellement infini) des jetons productibles, avec $\mathcal{Y} \subseteq \mathcal{M}$; c'est-à-dire des jetons initiaux, et des jetons non-initiaux produits par des nœuds de calculs sans place en entrée.*

$$\mathcal{Y} = \mathcal{I} \cup \{\, p_i \mid p_i \in \mathcal{M},\ \bullet\bullet p = \emptyset \}$$

Remarque 3.12. Un GDE est acyclique.

Remarque 3.13. Pour tout $p_i \in \mathcal{M}$, si le degré entrant de p_i est nul, alors $p_i \in \mathcal{Y}$.

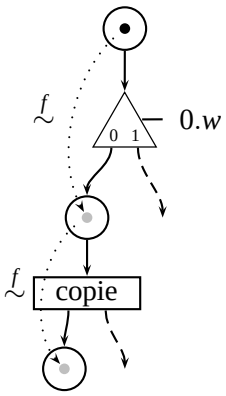
Remarque 3.14. Un GDE est infini si et seulement si son KRG correspondant est quasi-vivant.

Le GDE est utile pour se représenter le concept de graphe de dépendance. Cependant, le fait qu'il soit potentiellement infini le rend inapproprié pour être étudié plus en détails. Ainsi, nous souhaiterions définir une forme *finie* de graphe de dépendance. Pour cela, nous allons définir des notions d'*équivalence* entre jetons, de façon à limiter notre étude à un sous-ensemble fini : deux jetons équivalents sont «fusionnés» en un seul.

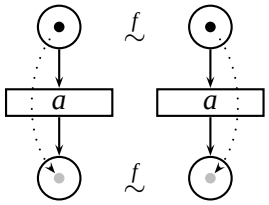
3.2.4 Première réduction : équivalence de flot

Nous définissons tout d'abord une première relation d'équivalence, dite *équivalence de flot*, et notée $\overset{f}{\sim}$. Deux jetons sont équivalents de flot si :

- l'un est issu du *routage* de l'autre (sélection, fusion, copie), mais n'a pas subi de transformation de la donnée qu'il transporte (pas de calcul) ;
- *ou* s'ils sont consommés par des nœuds qui font le même traitement, et que leurs résultats sont eux-mêmes équivalents de flot.



(a) Premier point de l'équivalence de flot : routage d'un jeton sans modifier la donnée.



(b) Second point de l'équivalence de flot : si les résultats sont équivalents et si les opérations sont équivalentes, alors les opérandes sont eux-mêmes équivalents, à condition qu'ils ne soient pas tous deux initiaux.

FIG. 3.6: Équivalence de flot.

Seuls les nœuds de calcul modifient les données des jetons ; les autres nœuds ne font que dupliquer ou router l'information. Aussi, peu importe le nombre de copies ou routages que subit une donnée entre deux calculs, nous considérons toutes ses instances équivalentes. Ceci correspond au premier point (*cf.* FIG. 3.6(a)). Pour le second point (*cf.* FIG. 3.6(b)), nous considérons deux nœuds de calcul de la même fonctionnalité, y compris les mêmes nombres d'entrées et de sorties. Si leurs résultats sont équivalents deux à deux, alors il en est de même pour leurs opérandes. Cependant, ces opérandes ne doivent pas être tous les deux des jetons initiaux : nous avons supposé dans la section précédente que le marquage initial ne pouvait être altéré, donc deux jetons initiaux sont différents par principe. Notons que l'équivalence de flot s'applique aussi bien en largeur qu'en profondeur : $p'_{i'}$ peut très bien être un successeur de p_i, tout comme un jeton d'un flot en parallèle.

Plus formellement, nous voudrions utiliser la définition suivante : pour tous $p_i, p'_{i'} \in \mathcal{M}$, $p_i \overset{f}{\sim} p'_{i'}$, si et seulement si :

1. Il existe un chemin dans le GDE de p_i à $p'_{i'}$, tels que toutes les dépendances de σ sont des dépendances de flot induites par des tirages de nœuds de routage ou de copie, *ou*
2. $p_i \xrightarrow{n}_f q_j \Leftrightarrow p'_{i'} \xrightarrow{n'}_f q'_{j'}$ et $\lambda(n) = \lambda(n')$ et $q_j \overset{f}{\sim} q'_{j'}$ et $p_i, p'_{i'} \notin \mathcal{I}$.

Cependant, il existe beaucoup de relations vérifiant cet énoncé, ce n'est donc pas une définition. Nous voulons choisir parmi toutes ces relations la plus «large». Nous procédons alors par étapes :

Définition 3.15. *Soit $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$ un GDE, et $\mathcal{R} \in \mathcal{M} \times \mathcal{M}$ une relation sur ses nœuds. Pour tout jetons $p_i, p'_{i'} \in \mathcal{M}$, si $p_i \mathcal{R} p'_{i'}$, alors l'un des points suivants est vérifié :*

1. *Il existe un chemin σ dans le GDE entre p_i et $p'_{i'}$, tel que toutes les arcs (dépendances) de σ sont des dépendances de flot, induites par des nœuds de routage, ou*
2. $p_i \xrightarrow{n}_f q_j \Leftrightarrow p'_{i'} \xrightarrow{n'}_f q'_{j'}$ *et* $\lambda(n) = \lambda(n')$ *et* $q_j \mathcal{R} q'_{j'}$ *et* $p_i, p'_{i'} \notin \mathcal{I}$.

Proposition 3.16. *Soit $\mathcal{R}_n$ une relation vérifiant la définition 3.15. Les relations suivantes vérifient aussi la définition 3.15 :*

1. $id_{\mathcal{M}}$
2. $\mathcal{R}_n^{-1}$

Posons la relation $\overset{f}{\sim}$ comme la clôture transitive des relations $\mathcal{R}_n$ vérifiant la définition 3.15. Pour tous jetons $p_i, p'_{i'} \in \mathcal{M}$, p_i et $p'_{i'}$ sont dits *équivalents de flot* si et seulement si $p_i \overset{f}{\sim} p'_{i'}$.

Proposition 3.17. *$\overset{f}{\sim}$ est une relation d'équivalence.*

La relation $\overset{f}{\sim}$ est définie comme une implication ; il ne reste plus qu'à vérifier la réciproque pour nous assurer que $\overset{f}{\sim}$ est bien la relation que nous cherchons à définir.

Proposition 3.18 (Équivalence de flot)**.** *Soit $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$ un GDE. Pour tous jetons $p_i, p'_{i'} \in \mathcal{M}$, p_i et $p'_{i'}$ équivalents de flot, et notés $p_i \overset{f}{\sim} p'_{i'}$, si et seulement si l'une des conditions suivantes est vérifiée :*

1. *Il existe un chemin σ dans le GDE de p_i à $p'_{i'}$, tels que tous ses arcs (dépendances) σ sont des dépendances de flot induites par des tirages de nœuds de routage ou de copie, ou*
2. $p_i \xrightarrow{n}_f q_j \Leftrightarrow p'_{i'} \xrightarrow{n'}_f q'_{j'}$ *et* $\lambda(n) = \lambda(n')$ *et* $q_j \overset{f}{\sim} q'_{j'}$ *et* $p_i, p'_{i'} \notin \mathcal{I}$.

Maintenant la relation d'équivalence définie, nous prouvons l'existence d'une forme finie du graphe de dépendance.

Lemme 3.19. *Soit $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ un KRG borné, avec une séquence de routage de partie initiale vide. Autrement dit :*

$$\forall n \in \mathcal{N}_s \cup \mathcal{N}_f,\ \exists v_n \in \mathbb{B}^*,\ R(n) = v_n^{\omega}$$

Alors :

1. *Pour tout $n \in \mathcal{N}_s \cup \mathcal{N}_f$, il existe $j_n \in \mathbb{N}^*$, tel que pour tout $k \in \mathbb{N}$, $R(n) = \mathrm{suf}(R(n), kj_n)$*
2. *Il existe une séquence de tirages au cours de laquelle chaque n est tiré exactement j_n fois.*

Proposition 3.20. *Soit $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ un KRG borné, et $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$ son GDE infini. Il existe $l, j \in \mathbb{N}$ $(j > 0)$ tels que :*

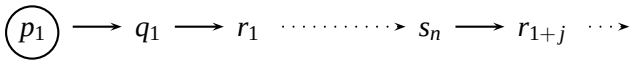
$$\forall p \in \mathcal{P},\ \forall i, k,\ i < j,\ p_{l+i} \overset{f}{\sim} p_{l+i+kj}$$

La proposition 3.20 nous donne une condition nécessaire et suffisante pour borner l'étude des dépendances : le comportement du graphe est ultimement périodique, après une phase initiale de longueur l. Au delà, tout jeton est équivalent à un jeton de la première période, de longueur j.

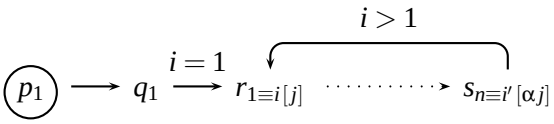
Nous sommes désormais en mesure de construire des graphes de dépendance $\overset{f}{\sim}$-réduits à partir de n'importe quel GDE : les nœuds équivalents du GDE sont regroupés en un seul dans le GDR.

Définition 3.21 (Graphe de dépendance $\overset{f}{\sim}$-réduit)**.** *Soit $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$ un GDE. Le $\overset{f}{\sim}$-GDR correspondant $\langle \mathcal{M}', \mathcal{D}', \mathcal{I}', \mathcal{Y}' \rangle$ est un GDE tel que :*

- *$\mathcal{M}'$ et $\mathcal{D}'$ sont tous deux finis.*
- *à tout sous-ensemble de jetons $\mathcal{M}'' \subseteq \mathcal{M}$, tel que tous les jetons de $\mathcal{M}''$ sont deux à deux équivalents, on fait correspondre un jeton dans $\mathcal{M}'$.*
- *pour tout couple de jetons de $\mathcal{M}'$, il existe une dépendance de flot entre ces jetons s'il existe une telle dépendance entre des jetons de leurs sous-ensembles correspondants dans $\mathcal{M}$.*
- *pour tout $p'_{i'} \in \mathcal{M}'$,*
  - *s'il existe $q'_{j'} \in \mathcal{M}'$, tel que $(q'_{j'}, p'_{i'}) \in \mathcal{D}'$, et $q'_{j'}$ a une infinité d'antécédents dans $\mathcal{M}$, et*
  - *s'il existe $r'_{k'} \in \mathcal{M}'$, tel que $(r'_{k'}, p'_{i'}) \in \mathcal{D}'$, et $r'_{k'}$ a un et un seul antécédent $r_k \in \mathcal{M}$,*



(a) Sous-graphe infini d'un GDE.



(b) GDR correspondant : repliage de la partie périodique.

FIG. 3.7: Réduction d'un GDE en GDR.

alors on attribue un fanion de validité aux dépendances. Soit p_i un antécédent de $p'_{i'}$, où $(r_k, p_i) \in \mathcal{D}$. $(r'_{k'}, p'_{i'})$ est valide si et seulement si $i' = i$, et $(q'_{j'}, p'_{i'})$ est valide si et seulement si $i' > i$.

– *$\mathcal{I}'$ et $\mathcal{Y}'$ sont les images de $\mathcal{I}$ et $\mathcal{Y}$ dans le GDR.*

La réduction de GDE en $\overset{f}{\sim}$-GDR est représentée en FIG. 3.7. Par exemple, tous les jetons numérotés r_{1+kj} sont fusionnés en un seul nœud $r_{1\equiv i\,[j]}$. Sans l'ajout de fanions, la réduction ajouterait des dépendances supplémentaires à $r_{1\equiv i\,[j]}$, qui dépendrait à la fois de q_1 et de $s_{n\equiv i'\,[\alpha j]}$. Pour cette raison, nous ajoutons des assertions exclusives sur les arcs entrants : dans le premier cas, la dépendance est valide si et seulement si $i = 1$; sinon, elle est valide si et seulement si $i > 1$. Notons que le $\overset{f}{\sim}$-GDR ne tient compte que des dépendances de flot : les dépendances de sortie ne sont pas prises en compte par la relation $\overset{f}{\sim}$; elles sont abstraites, et par conséquent les anti-dépendances également.

Exemple 3.22. Revenons à l'exemple que nous avions donné en FIG. 3.3. Nous voulions montrer une «équivalence» entre les graphes 3.3(a) et 3.3(b). Les dépendances de flot de leurs GDE respectifs sont représentées en FIG. 3.8(a) et 3.8(c). En étudiant leurs équivalences de flot, nous constatons que leurs $\overset{f}{\sim}$-GDR sont effectivement égaux (*cf.* FIG. 3.8(b) et 3.8(d)).

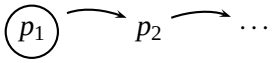
En revanche, le graphe 3.3(c) ne devait pas être équivalent : il présente un verrou. Les dépendances de flot de son GDE sont représentées en FIG. 3.8(e). Effectivement, nous constatons que son $\overset{f}{\sim}$-GDR, en FIG. 3.8(f), est différent des deux autres. Par ailleurs, remarquons que la situation de verrou y apparaît de façon évidente : il existe une boucle sur p_2, alors que ce jeton n'est pas initial.

3.2.5 Seconde réduction : équivalence de comportement

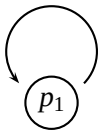
La relation $\overset{f}{\sim}$ ne prend pas en compte les dépendances de sortie. Nous souhaiterions une notion plus stricte quant à l'ordre de passage des jetons. Nous définissons alors une notion de *précédence* entre jetons, puis une nouvelle relation d'équivalence, dite *équivalence de comportement*, et notée $\overset{b}{\sim}$.

Définition 3.23 (Précédence)**.** *Soit $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R\rangle$ un KRG borné, et $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y}\rangle$ son GDE. Pour tous jetons $p_i, p'_{i'} \in \mathcal{M}$, p_i précède $p'_{i'}$ si et seulement si $p_i = p'_{i'}$, ou s'il existe $q_j, q_{j'}$, $j < j'$ tels que :*

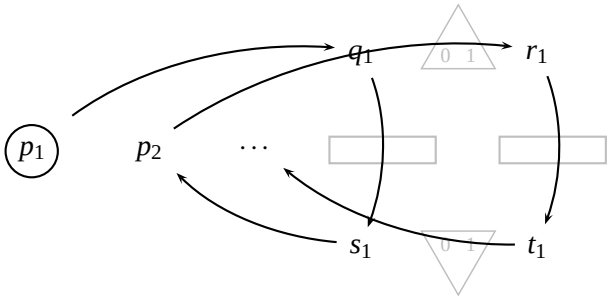
1. $\exists n, \ldots, n' \in \mathcal{N}_c \cup \mathcal{N}_f \cup \mathcal{N}_s$, $p_i \xrightarrow{n}_f \circ \cdots \circ \xrightarrow{n'}_f q_j$, *et*
2. $\exists n, \ldots, n' \in \mathcal{N}_c \cup \mathcal{N}_f \cup \mathcal{N}_s$, $p'_{i'} \xrightarrow{n}_f \circ \cdots \circ \xrightarrow{n'}_f q_{j'}$



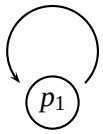
(a) Dépendances de flot du KRG de la FIG. 3.3(a).



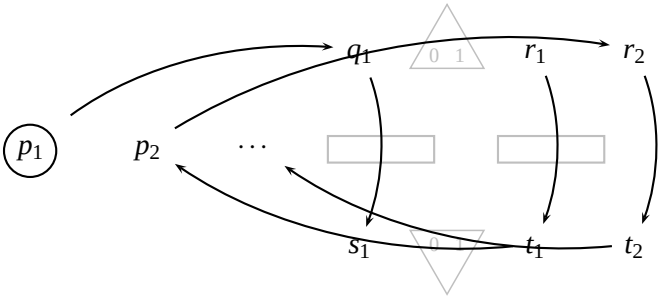
(b) $\overset{f}{\sim}$-GDR du KRG de la FIG. 3.3(a).



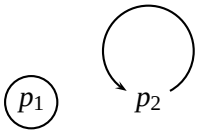
(c) Dépendances de flot du KRG de la FIG. 3.3(b).



(d) $\overset{f}{\sim}$-GDR du KRG de la FIG. 3.3(b).



(e) Dépendances de flot du KRG de la FIG. 3.3(c).



(f) $\overset{f}{\sim}$-GDR du KRG de la FIG. 3.3(c).

FIG. 3.8: Exemples de dépendances de flot et de leur réduction.

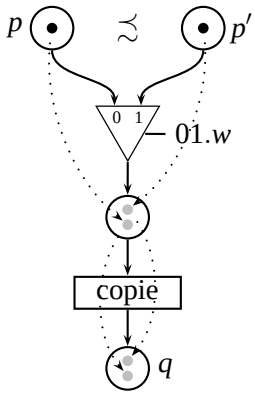


FIG. 3.9: Précédence entre jetons.

Définition 3.24. *Soit $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ un KRG borné, et $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$ son GDE. Nous posons la relation $\precsim$ comme la clôture transitive de l'ensemble des relations de précédence, selon la Définition 3.23.*

Avec les notations de la définition 3.23, on peut alors écrire $p_i \precsim_{q_j} p'_{i'}$ pour préciser que la précédence entre p_i et $p'_{i'}$ est établie via q_j.

Proposition 3.25. *$\precsim$ est une relation de pré-ordre.*

Exemple 3.26. La FIG. 3.9 illustre la relation de précédence entre jetons. *A priori*, les jetons des places p et p' sont indépendants. Pourtant, ils permettent de produire des jetons dans une place q, qui sont eux-mêmes en séquence (dépendance de sortie). La relation de précédence permet donc de faire remonter cette information aux jetons en amont.

Maintenant qu'une notion de précédence est établie, nous pouvons définir la relation d'équivalence de comportement comme un raffinement de l'équivalence de flot.

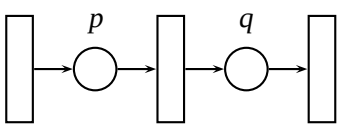
Définition 3.27 (Équivalence de comportement)**.** *Soit $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ un KRG borné, et $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$ son GDE. Pour tous jetons $p_i, p'_{i'} \in \mathcal{M}$, nous définissions l'équivalence de comportement, notée $p_i \overset{b}{\sim} p'_{i'}$, comme la clôture transitive de la relation :*

- *$p_i \overset{f}{\sim} p'_{i'}$, et*
- *soit $p = p'$,*
- *soit il n'existe pas de jeton q_j, avec $q_j \overset{f}{\not\sim} p_i$ et $q_j \overset{f}{\not\sim} p'_{i'}$, tel que q_j soit relié à p_i ou $p'_{i'}$ par un chemin de dépendances de sortie.*

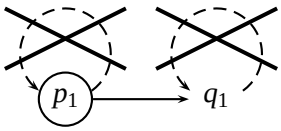
Ce que nous dit cette relation, c'est qu'outre le fait d'être équivalents de flot, deux jetons sont équivalents de comportement s'ils n'existe pas d'autre jeton non-équivalent avec lequel ils puissent être en relation de précédence. Cette condition permet de de s'assurer que p_i et $p'_{i'}$ sont suffisamment «indépendants» des autres jetons pour pouvoir être fusionnés par la suite. Ce point est important à noter, car il nous permet de prendre en compte les ordres de passage. La condition sur $p = p'$ permet quant à elle de conserver la propriété de périodicité de la proposition 3.20.

Proposition 3.28. *$\overset{b}{\sim}$ est une relation d'équivalence.*

Proposition 3.29. *Pour tous jetons $p_i, q_j \in \mathcal{M}$, si $p_i \overset{b}{\sim} q_j$, alors $p_i \overset{f}{\sim} q_j$.*



(a) Exemple de KRG vivant.



(b) $\overset{b}{\sim}$-GDR correspondant au KRG (a). Les boucles formées par les dépendances de sorties, issues de la fusions des jetons de p et q respectivement, sont supprimées.

FIG. 3.10: Exemple de suppression de dépendances de sortie.

Le lecteur notera que l'on peut faire le parallèle avec les travaux de Milner sur les équivalences et bisimulations en CCS [167, 168]. En effet, ces relations d'équivalence sont des bisimulations au sens de Milner ; bien que les modèles soient différents, les raisonnements et buts poursuivis sont très similaires.

La définition 3.21 doit être adaptée en conséquence pour les $\overset{b}{\sim}$-GDR. Les deux seules différences sont, d'une part, qu'un $\overset{b}{\sim}$-GDR est réduit selon la relation $\overset{b}{\sim}$, et d'autre part, qu'un $\overset{b}{\sim}$-GDR contient des dépendances de sortie et des anti-dépendances : contrairement au $\overset{f}{\sim}$-GDR, ces deux types de dépendances n'y sont pas abstraites. Cependant, *toutes* les dépendances de sorties et anti-dépendances *ne sont pas présentes* dans le $\overset{b}{\sim}$-GDR : en effet nous supprimons (i) les boucles issues de la fusion de plusieurs jetons en un seul, et (ii) pour les jetons d'un même flot, les dépendances du dernier jeton de la période vers le premier. Le $\overset{b}{\sim}$-GDR intègre les dépendances de sortie et anti-dépendances pour représenter la séquentialité des opérations au sein d'une période. Il faut néanmoins garder à l'esprit que le $\overset{b}{\sim}$-GDR n'est qu'une *abstraction* du comportement du KRG, et conserver les dépendances de sortie et les anti-dépendances d'une période à l'autre reviendrait à surcontraindre le modèle en introduisant des circuits sans jeton productible.

Exemple 3.30. Nous illustrons le principe en FIG. 3.10. Le KRG de la FIG. 3.10(a) est vivant, il peut s'exécuter à l'infini. Les places p et q sont potentiellement traversées par une infinité de jetons ; cependant tous les jetons traversant p sont équivalents de comportement, de même pour les jetons traversant q. Leur abstraction en p_1 et q_1, respectivement, ferait *a priori* reboucler les dépendances de sorties (*cf.* FIG. 3.10(b)). En particulier, la boucle sur q_1 ne contiendrait pas de jeton productible ; comme nous ne verrons dans la sous-section suivante, cela aurait pour effet d'introduire un verrou dans la forme normale, alors que le KRG initial est vivant. Pour cette raison, ces boucles sont supprimées.

3.2.6 Forme normale

Nous pouvons enfin effectuer la transformation inverse et construire une forme normale du KRG à partir de son GDR. Le fait d'avoir expansé les dépendances de données jusqu'à la fin de la première période revient à dérouler complètement le routage dans la forme normale. Plus précisément, le KRG normalisé consiste en une partie initiale et acyclique, reliée à une partie périodique et cyclique par des nœuds de fusion. Ces fusions ont une particularité : seul le premier jeton est consommé sur l'entrée «initiale», les suivants étant consommés infiniment souvent sur l'entrée «périodique». Cela correspond bien à un quasi-graphe marqué (*cf.* définition 3.10).

La construction de la forme normale est détaillée par l'algorithme 3.1. La FIG. 3.11 illustre la construction de la forme normale d'un KRG, en passant par son $\overset{b}{\sim}$-GDR.

Entrées : un graphe de dépendance réduit $\langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$ privé de ses dépendances de sortie.
Sorties : un quasi-graphe marqué $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$.

pour tout jeton p_i dans $\mathcal{M}$ **faire**
    Étape 1 : construire une place pour chaque jeton du GDR et son marquage initial
    construire une place $place(p_i)$ (*cf.* FIG. 3.12(a) et (b))
    si p_i est productible **alors** le marquage de $place(p_i)$ vaut 1 **sinon** il vaut 0

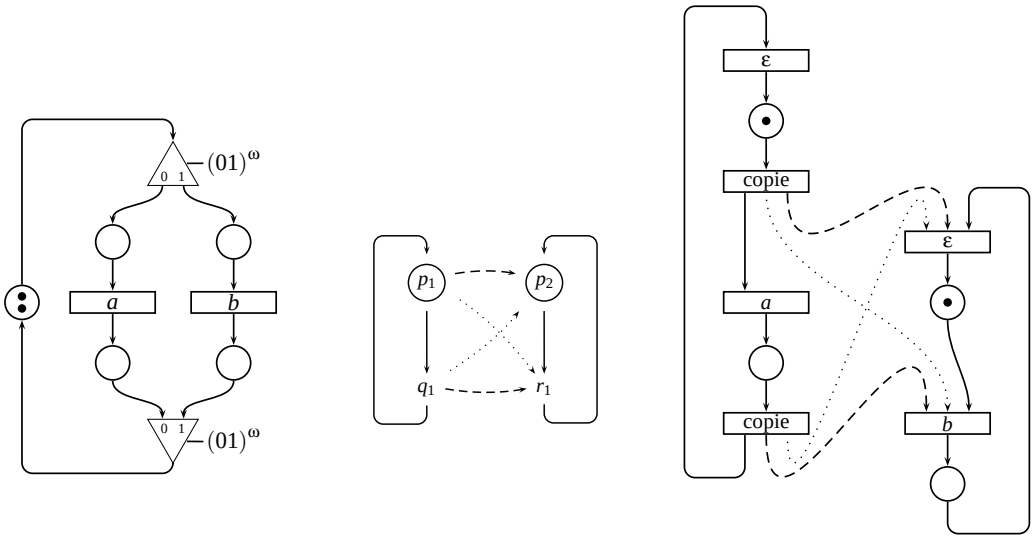
    Étape 2 : construction du nœud en entrée
    construire un nœud de calcul n et le lier à $place(p_i)$
    si p_i a une dépendance de flot entrante, induite par un nœud de label l **alors**
        $\lambda(n) \leftarrow l$ (*cf.* FIG. 3.12(e) et (f))
    sinon
        $\lambda(n) \leftarrow \varepsilon$
        si le degré entrant de p_i nul **alors**
            ajouter une boucle sans marquage initial sur n (*cf.* FIG. 3.12(c) et (d))
        fin si
    fin si

    Étape 3 : insertion des nœuds de copie
    si p_i a plusieurs dépendances sortantes **alors**
        construire un nœud de copie et le lier à $place(p_i)$ (*cf.* FIG. 3.12(i) et (j))
    fin si

    Étape 4 : insertion des nœuds de fusion
    si p_i a deux dépendances entrantes exclusives, selon des fanions de validité, **alors**
        construire un nœud de fusion et les lier à n (*cf.* FIG. 3.12(g) et (h))
    fin si
fin pour

Étape 5 : finalisation de l'interconnexion
pour tout dépendance dans $\mathcal{D}$ **faire**
    relier les nœuds et places correspondantes de façon appropriée
fin pour

Algorithme 3.1: Construction du quasi-graphe marqué.



(a) Exemple de KRG.

(b) $\overset{b}{\sim}$-GDR. Les dépendances de sorties sont en pointillés longs, et les anti-dépendances en pointillés courts.

(c) Forme normale (quasi-graphe marqué) totalement périodique. Par soucis de simplicité, seules les quatre places correspondant aux quatre jetons du $\overset{b}{\sim}$-GDR sont représentées.

FIG. 3.11: Exemple de KRG et de sa forme normale. Les routages du KRG sont périodiques, sa forme normale n'a donc pas de partie initiale.

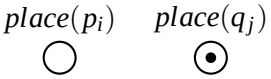
Théorème 3.31 (Existance d'une forme normale)**.** *À tout KRG correspond une unique forme normale, construite par l'algorithme 3.1.*

Remarque 3.32. Un point important à prendre en compte est qu'une forme normale KRG peut contenir autant ou *plus* de jetons que le KRG dont elle est issue ; jamais *moins*, il n'y a jamais perte de données. En effet, le marquage du quasi-graphe marqué construit par l'algorithme 3.1 dépend des jetons *productibles* du $\overset{b}{\sim}$-GDR, et non des jetons *initiaux*. Pourtant, cela ne signifie pas que nous ayons «créé» des données. La raison en est la suivante. Par la définition 3.11, un jeton productible n'a pas de dépendance entrante : il peut donc être vu comme une constante, ou du moins, comme le résultat d'une expression qui ne dépend pas des données. Ainsi, cet «ajout» de jetons est en fait une simple propagation de constantes. Cela ne modifie en rien la sémantique du KRG.

Par transformations successives, nous avons exprimé la sémantique d'un KRG selon celle, plus simple, de deux graphes marqués unis par des fusions. Construire les $\overset{f}{\sim}$- et $\overset{b}{\sim}$-quasi-graphes marqués d'un KRG quelconque nous donne un point de référence au niveau sémantique : deux KRG sont *équivalents* si et seulement si leurs formes normales sont identiques. Ce raisonnement est similaire à celui de l'expansion des graphes SDF en graphes marqués. Ainsi, des propriétés de vivacité peuvent être extraites directement d'un KRG, ou en appliquant des résultats bien connus des graphes marqués sur sa forme normale (*cf.* sous-section 2.2).

Théorème 3.33 (Vivacité d'un KRG)**.** *Un KRG est quasi-vivant si et seulement si son $\overset{b}{\sim}$-quasi-graphe marqué est quasi-vivant. Aussi, un KRG est vivant si et seulement si son $\overset{b}{\sim}$-quasi-graphe marqué est vivant.*

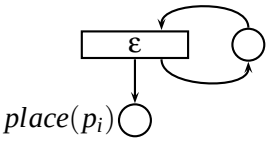
(a) État du jeton : productible $(q_j \in \mathcal{Y})$ ou non $(p_i \notin \mathcal{Y})$.



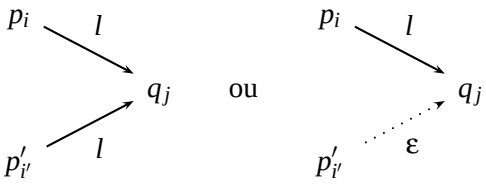
(b) Si le jeton est productible, le marquage de la place correspondante vaut 1.

$$d^-(p_i) = 0$$

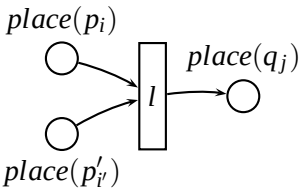
(c) Jeton sans dépendance entrante.



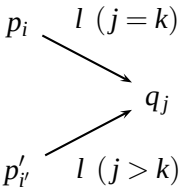
(d) Création d'un circuit sans marquage pour interdire des productions de jeton.



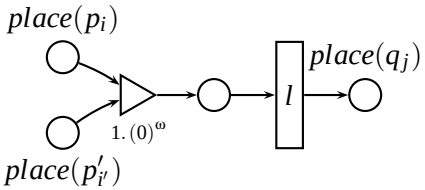
(e) Dépendances vers q_j d'un ou plusieurs jetons. Les dépendances de flot sont induites par des nœuds de label l, les autres sont étiquetées ε.



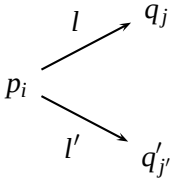
(f) Création d'un nœud, de label l, consommant p_i et $p'_{i'}$ et produisant q_j.



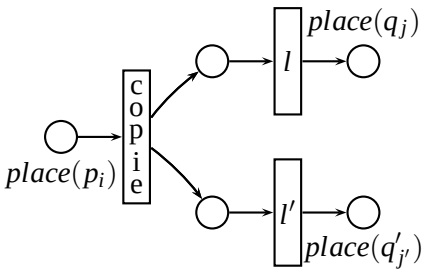
(g) Dépendances avec un fanion de validité.



(h) Insertion d'un nœud de fusion intermédiaire et d'une nouvelle place pour réunir les flots : initial et périodique.



(i) Dépendance d'un jeton p_i vers plusieurs jetons.



(j) Insersion d'un nœud de copie pour dupliquer le flot, et routage vers les différents consommateurs.

FIG. 3.12: Éléments de construction d'un quasi-graphe marqué *(droite)* à partir d'un GDR *(gauche)*.

3.3 Transformation de l'interconnexion

Dans cette section, nous décrivons un ensemble «d'identités remarquables», qui nous permettent de transformer la topologie de l'interconnexion des nœuds de calcul. Nous prouvons que ces transformations sont *correctes*, c'est-à-dire qu'elles préservent la fonctionnalité de l'application modélisée. Par fonctionnalité, on entend les relations entre flots de jetons entrant et sortant d'une part, et préservation de la vivacité et de l'absence de blocage.

Pour cela, nous montrons que pour tout KRG, son interconnexion peut s'exprimer sous forme *expansée* : informellement, cela revient à décroiser et départager les chemins entre tout couple producteur/consommateur. Si nous construisons un second KRG en appliquant successivement ces identités remarquables, nous prouvons que ces deux KRG ont la même forme expansée. Ces transformations de l'interconnexion s'articulent en quatre étapes :

1. La définition et la preuve de correction des identités remarquables ;
2. La définition d'arbres de sélection et de fusion et leur équilibrage : minimiser le nombre maximum de nœuds de sélection ou de fusion en cascade que doit traverser un jeton avant de sortir de l'arbre ;
3. La gestion des permutations de jetons entre leur producteur et leur consommateur : nous construisons un sous-graphe dédié aux dépassements de jetons ;
4. L'expansion de l'interconnexion : construction d'un réseau de liens point-à-point.

3.3.1 Transformations locales

Des identités remarquables, présentées intuitivement par Boucaron [33], permettent de modifier la topologie du graphe en utilisant les propriétés des *sélections* et *fusions*, tout en respectant la fonctionnalité du graphe : autrement dit, ces transfornations doivent garantir que pour tout couple producteur/consommateur, les relations entre flots sortants et entrants sont respectées. Des jetons peuvent être aiguillés sur des chemins distincts, puis être réunis dans un même flot, tout en conservant leur ordre d'origine.

Expansion d'une place

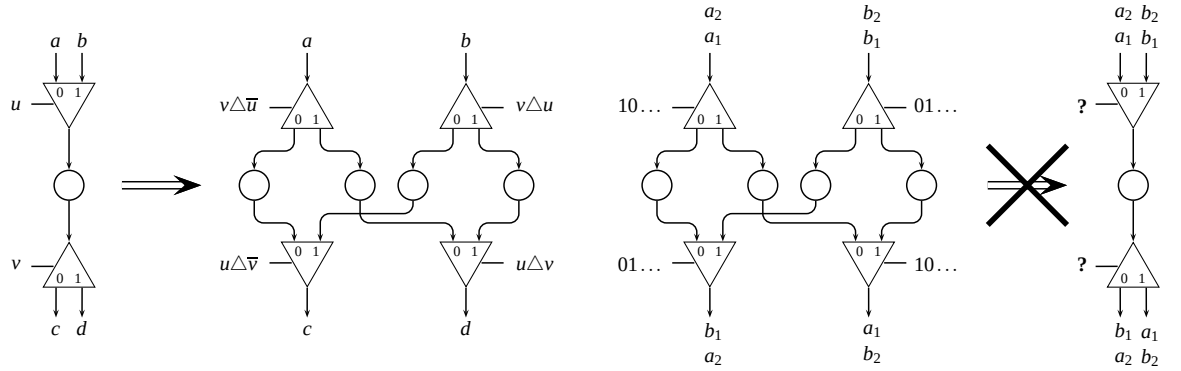
Par exemple, une place peut être partagée entre plusieurs flots ; cela peut être, de façon abstraite, les signaux de différents modules transitant par un bus de données commun. L'expansion d'une telle place, représentée en FIG. 3.13(a), revient à remplacer un médium de communication et de stockage partagé par autant de liaisons point à point que de flots y transitant. Cette transformation introduit davantage de concurrence, mais préserve les ordres des jetons au sein de chaque flot.

Lemme 3.34. $\forall u, v \in \mathbb{P}, \forall i \in \mathbb{N}^*$,

$$v_{[u]_i} = 1 \;\Rightarrow\; |\mathrm{prf}(v, \mathrm{idx}_1(u,i))|_1 = \mathrm{idx}_1(u \triangle v, |\mathrm{prf}(v \triangle u, i)|_1) \tag{3.19}$$

Proposition 3.35 (Expansion d'une place)**.** *L'expansion d'une place (ExpPlace), telle que représentée en* FIG. 3.13(a)*, préserve les relations d'ordre sur les flots de jetons.*

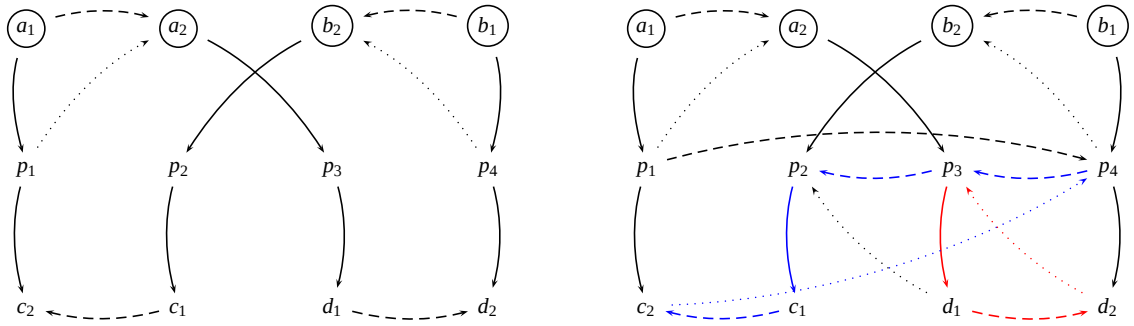
Notons que la réciproque n'est pas toujours possible : partant d'une topologie expansée, il n'existe pas forcément de séquence de routage permettant d'effectuer la transformation inverse, et obtenir une forme factorisée (*FactPlace*, *cf.* FIG. 3.13(b)). Ce en raison que certains ordres de jetons, en entrée comme en sortie, et compatibles avec des liaisons point-à-point, ne peuvent être reproduits avec une place partagée : la factorisation impose une séquentialisation des jetons, donc un ordre total.



(a) Expansion d'une place *(ExpPlace)*.

(b) Exemple de graphe où la factorisation *(FactPlace)* est impossible.

FIG. 3.13: Expansion et factorisation d'une place.



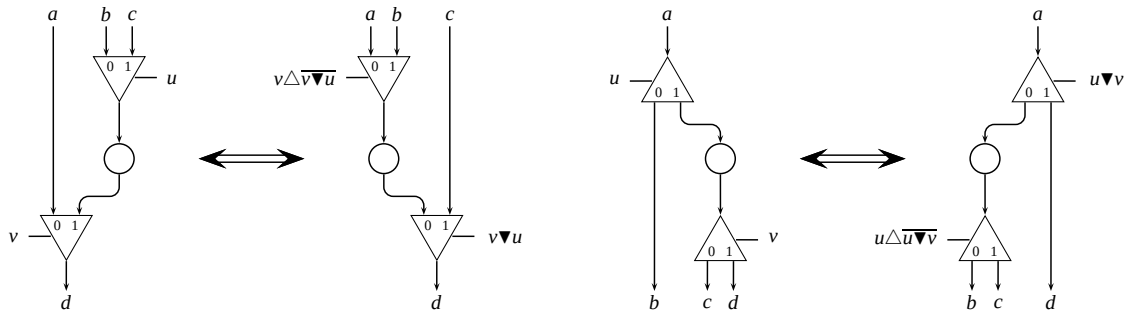
(a) Dépendances entre jetons du sous-graphe sous sa forme expansée, en FIG. 3.13(b) *(gauche)*.

(b) Dépendances entre jetons du sous-graphe, factorisé arbitrairement, en FIG. 3.13(b) *(droite)*. La séquentialisation $a_1b_1a_2b_2$ introduit des verrous. Deux circuits sans jetons initiaux sont représentés en rouge et bleu.

FIG. 3.14: Comparaison des dépendances entre forme expansée et factorisée : introduction de dépendances lors de la séquentialisation.

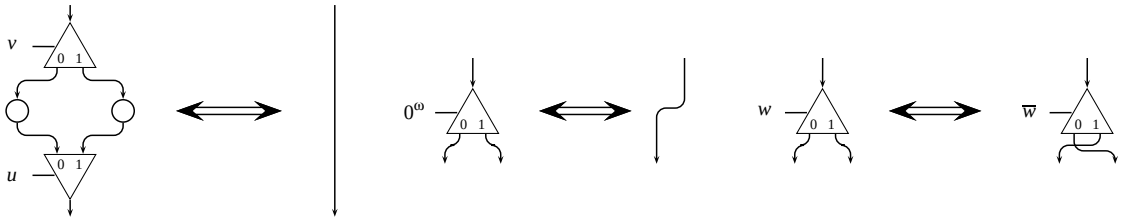
Exemple 3.36. La FIG. 3.14 présente un exemple de factorisation, avec séquentialisation des jetons suivant l'ordre $a_1b_1a_2b_2$. Dans le cas parallèle, aucune contrainte n'est exercée sur les jetons traversant les places intermédiaires (p_1 à p_4, *cf.* FIG. 3.14(a)). Le sous-graphe de dépendance étendu reste acyclique.

En revanche, dans le cas factorisé, nous avons contraint l'ordre de passage des jetons dans la place intermédiaire. L'ajout de dépendances de sorties introduit des anti-dépendances supplémentaires (*cf.* FIG. 3.14(b)). Nous introduisons alors plusieurs circuits, dont certains ne contiennent pas de jetons initiaux : si a_2 et b_2 entrent en second, ils ne peuvent sortir en premier. Quel que soit l'ordre dans lequel les jetons sont consommés par la fusion, il est impossible qu'ils soient produits par la sélection dans l'ordre imposé. Dans ce cas, la factorisation est donc impossible.



(a) Permutation de nœuds de fusion *(PermFusion)*. (b) Permutation de nœuds de sélection *(PermSelect)*.

FIG. 3.15: Permutations des nœuds de routage.



(a) (b) (c)

FIG. 3.16: Exemples de simplifications du routage.

Permutations de sélections et fusions

D'autres transformations nous permettent de modifier localement la topologie du graphe en permutant des nœuds de routage, comme représenté en FIG. 3.15. Ces deux transformations permettent de réaiguiller localement des jetons.

Proposition 3.37 (Permutation de fusions)**.** *La permutation de nœuds de fusion* (PermFusion)*, telle que représentée en* FIG. 3.15(a)*, préserve les relations d'ordre sur les flots de jetons.*

Proposition 3.38 (Permutation de sélections)**.** *La permutation de nœuds de sélection* (PermSelect)*, telle que représentée en* FIG. 3.15(b)*, préserve les relations d'ordre sur les flots de jetons.*

Correction et simplification des transformations

Enfin, maintenant que la préservation des flots de jetons est établie, nous pouvons montrer un critère de correction de nos transformations : la (quasi)-vivacité du KRG est préservée par ces transformations, et elles n'affectent pas les situations de bloquage.

Corollaire 3.39 (Consistance des transformations)**.** *Les transformations d'expansion des places* ExpPlace *(*cf. FIG. 3.13(a)*), de permutation des fusions* PermFusion *(*cf. FIG. 3.15(a)*), et de permutations des sélections* PermSelect *(*FIG. 3.15(b)*) n'altèrent pas le caractère borné du graphe, ni ses propriétés de vivacité ou d'absence de blocage.*

Aussi, ces transformations peuvent amener à certaines simplifications de la topologie, que nous décrivons par les remarques suivantes.

Remarque 3.40. Le motif de la FIG. 3.16(a) est équivalent à un arc si et seulement si les routages de ses nœuds sont égaux : $u = v$. Dans ce cas, les jetons sortent dans le même ordre qu'ils sont entrés.

De plus, un nœud de sélection ou de fusion dont la séquence de routage vaut 0^ω ou 1^ω, comme en FIG. 3.16(b), est aussi équivalent à un arc : les jetons sont toujours routés *vers* ou *depuis* la même place, l'autre branche est inatteignable. Elle représente, en quelque sorte, du code mort.

Remarque 3.41. Les entrées et sorties, respectivement, de nœuds de fusion et de sélection sont interchangeables en remplaçant leurs séquences de routages par leurs négations (*cf.* FIG. 3.16(c)).

3.3.2 Arbres de sélection et de fusion

Comme nous l'avons mentionné précédemment, les nœuds de fusion et de sélection correspondent respectivement à des multiplexeurs 2-vers-1 et démultiplexeurs 1-vers-2. Pour des routages plus complexes, il est alors nécessaire de mettre ces nœuds en cascade, afin de représenter du routage *n*-vers-1 et 1-vers-*n*. Par exemple, la fusion de *n* flots de jetons en un seul est modélisée par un arbre de nœuds de fusion. Nous généralisons alors le routage à *n* entrées ou sorties au moyen de «macros», que nous nommons *nœuds de n-sélection* et *n-fusion*.

n-Sélection

Définition 3.42 (*n-Sélection*)**.** *Un nœud de n-sélection est composé d'un arbre de* $n-1$ *nœuds de sélection, à n sorties. Par analogie avec les nœuds de sélection, sa condition de branchement est une séquence ultimement périodique sur* $[\![0, n-1]\!]$. *Les sorties du nœud de n-sélection sont numérotées de* 0 *à* $n-1$.

Le tirage d'un nœud de *n*-sélection est atomique : il correspond au tirage de toutes les sélections, de la racine (entrée) à une feuille (sortie), sur le chemin suivi par le jeton. Vu de l'extérieur, son comportement est semblable à celui des nœuds de sélection binaires ; en interne, le routage des jetons est déterminé par l'arbre de sélection sous-jacent.

La valeur de la séquence entière de routage du nœud de *n*-sélection détermine la valeur de l'ensemble des séquences binaires de routage, internes au nœud. La concordance entre la séquence entière et les séquences binaires qui lui sont subordonnées est la suivante :

- Le $i^{\text{ème}}$ élément de la séquence entière indique sur quelle sortie du nœud de *n*-sélection est aiguillé le $i^{\text{ème}}$ jeton en entrée. Ce nombre entier est lu en base 2, sous forme un mot binaire, dont les bits sont ordonnés du moins significatif au plus significatif ;
- Lors du tirage du nœud de *n*-sélection, la tête w (entier converti en mot binaire) de sa séquence de routage est consommée ;
- En interne, on consomme récursivement les bits de w ; ils indiquent, au fur et à mesure de la descente dans l'arbre, de quel côté doit être aiguillé le jeton (gauche ou droite, «0» ou «1») ;
- Les conditions d'arrêt de ce parcours sont :
  1. avoir complètement consommé w, on complète les séquences restantes par des «0» si nécessaire ;
  2. atteindre une feuille : la «queue» non consommée de w est une séquence de «0», que l'on ignore.

Exemple 3.43. La FIG. 3.17 illustre le principe du routage des arbres de sélection, et de la concordance entre routages entiers et binaires. Supposons que la tête du nœud de 4-sélection vaut $2_{10} = 10_2$;

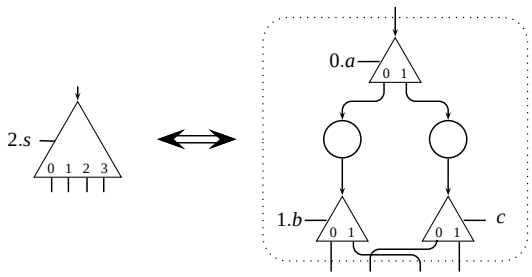


FIG. 3.17: Nœud de 4-sélection, et routage du premier jeton sur la sortie 2.

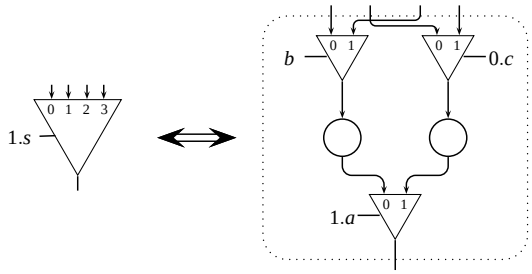


FIG. 3.18: Nœud de 4-fusion, et routage du premier jeton depuis l'entrée 1.

parmi toutes les sorties, numérotées de «0» à «3», c'est sur la «2» que sera produit un jeton au prochain tirage. Le bit le moins significatif vaut «0» ; la tête de la séquence de routage de la racine vaut donc «0». On descend alors du côté gauche. Le second bit vaut «1», donc la tête de la séquence de routage du nœud suivant vaut donc «1».

Cette façon d'associer des chemins à des nombres entiers au sein d'un arbre de *n*-sélection est équilibrée : au niveau de la racine, le sous-arbre de gauche correspond aux chemins pairs, le droit aux chemins impairs, et ce récursivement. Le sous-arbre de gauche contient, au plus, un nœud de plus que le sous-arbre droit. La hauteur d'un arbre à n sorties est alors égale à $\lceil \log_2(n) \rceil$.

n-Fusion

Exactement de la même manière que nous avons introduit les arbres de *n*-sélection, nous présentons maintenant les arbres de *n*-fusion. Le tirage d'un nœud de *n*-fusion est atomique, donc la lecture d'un nœud de *n*-fusion est bloquante

Définition 3.44 (*n-Fusion*)**.** *Un nœud de n-fusion est composé d'un arbre de* $n-1$ *nœuds de fusion, ayant n entrées. Par analogie avec les nœuds de fusion, sa séquence de routage est une séquence ultimement périodique sur* $[\![0, n-1]\!]$. *Les entrées du nœud de n-fusion sont numérotées de* 0 *à* $n-1$.

Exemple 3.45. La FIG. 3.18 repésente un exemple d'arbre de 4-fusion. L'entrée à lire est déterminée par la première lettre de sa séquence de routage, ici $1_2 = 1_{10}$. Partant de la racine, la tête de sa séquence de routage binaire est égale au bit le moins significatif, donc «1». Nous partons alors du côté droit. Nous complétons la séquence de routage du prochain nœud rencontré par «0» ; c'est une feuille, nous nous arrêtons.

De même qu'un nœud de *n*-sélection, un nœud de *n*-fusion contient $n-1$ nœuds de fusion, pour une hauteur maximale dans le cas idéal de $\lceil \log_2(n) \rceil$.

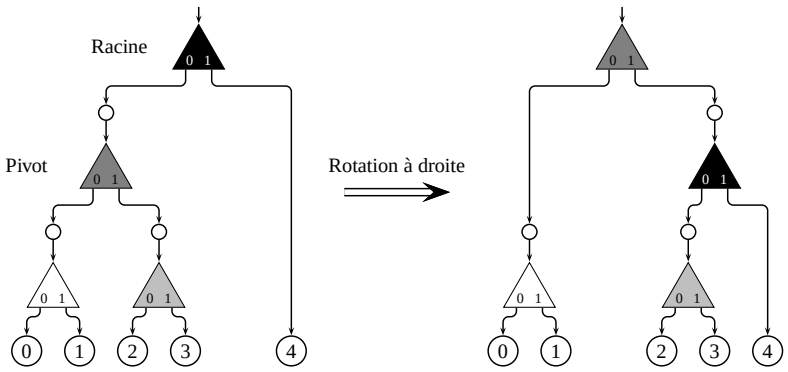


FIG. 3.19: Équilibrage d'un arbre de 5-sélection.

Équilibrage d'un arbre

Maintenant, considérons un arbre de fusion ou sélection non-balancé ; nous cherchons à l'équilibrer. Un tel arbre peut être vu comme un cas particulier d'arbres binaires de recherche, nous permettant ainsi d'utiliser les résultats bien connus sur le (ré)équilibrage des arbres AVL par rotations successives [1, 135] (*cf.* FIG. 3.19). Ces transformations ont justement été prouvées comme ne modifiant pas l'ordre des jetons (propositions 3.37 et 3.38), ni altérant le caractère borné du graphe (corollaire 3.39).

Théorème 3.46 (Adelson-Velskii et Landis)**.** *La hauteur h d'un arbre AVL avec n nœuds internes est bornée par :*

$$\log_2(n+1) \leq h < \log_\phi(n+2) + \log_\phi\left(\sqrt{5}\right) - 2 \tag{3.20}$$

Vu qu'une n-sélection (n feuilles) est réalisée par un arbre de $n-1$ nœuds de sélection ($n-1$ nœuds internes), on peut ainsi borner la hauteur d'un arbre de n-sélection entre $\log_2(n)$ et $\log_\phi(n+1) + \log_\phi\left(\sqrt{5}\right) - 2$, par application du théorème 3.46. Il en est de même pour les arbres de fusion.

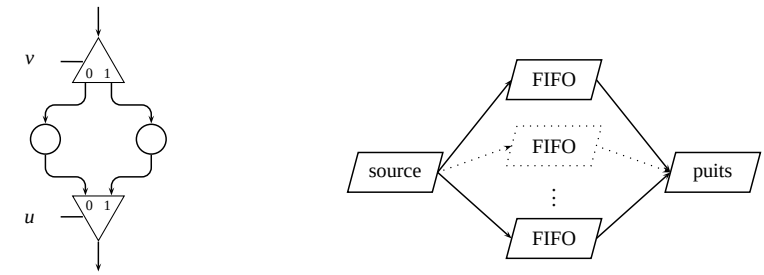
Notons que l'équilibrage des arbres rouge-noir repose également sur des rotations de nœud. Cependant, ces arbres ne garantissent qu'une hauteur maximale égale à $2\log_2 n$ dans le pire des cas [68, 113].

3.3.3 Permutations de jetons

Considérons deux nœuds de calcul, reliés par une quelconque interconnexion, tels que des jetons produits par l'un sont routés vers l'autre. Jusqu'à présent, nous n'avons considéré que des consommations de jetons dans l'ordre dans lequel ils sont produits. Si, au contraire, les jetons sont consommés dans un ordre différent, nous cherchons à construire un sous-graphe permettant de les permuter.

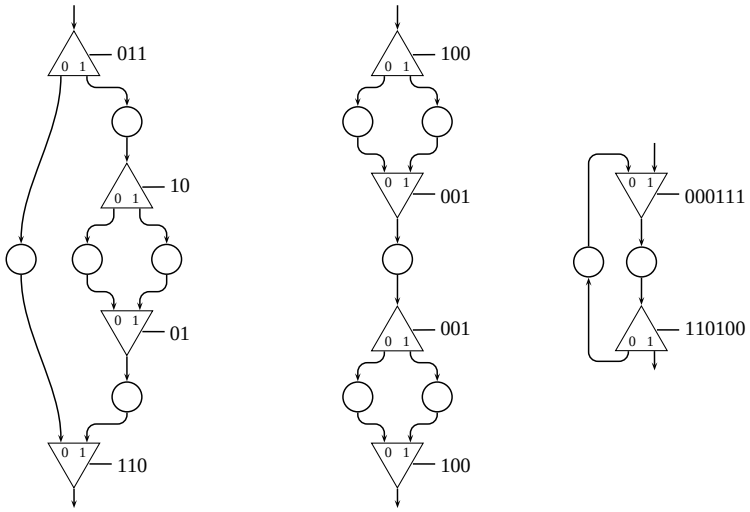
Comme nous l'avons vu, les jetons suivant un même chemin sont totalement ordonnés (*cf.* définition 3.6). Pour qu'un jeton puisse en dépasser un autre, ils doivent emprunter des chemins différents. Par conséquent, le problème de la permutation des jetons peut être résolu en séparant puis fusionnant des flots de jetons, tout en jouant sur leurs routages. Autrement dit, des jetons sont «parqués» dans une place, en attendant que leurs successeurs les dépassent. Le sous-graphe dédié à cette tâche est appelé un *trieur*.

Exemple 3.47. La FIG. 3.20(a) présente un trieur élémentaire. Les jetons orientés vers une place peuvent être dépassés par ceux routés sur l'autre place. Si nous posons $u = (011)^\omega$ et $v = (110)^\omega$, les premiers jetons de chaque période sont permutés avec leurs deux successeurs, de telle façon que le flot



(a) Trieur élémentaire. (b) Réseau de FIFOs en parallèle.

FIG. 3.20: Exemples de trieurs de jetons.



(a) Trieur parallèle. (b) Trieur série. (c) Trieur itératif.

FIG. 3.21: Différents trieurs réalisant la permutation $\pi = [3,2,1]$.

d'entrée $[1,2,3,4,5,6]$ produira le flot de sortie $[2,3,1,5,6,4]$. Si maintenant nous posons $u = v$, alors les jetons sortent dans l'ordre dans lequel ils sont entrés : le trieur effectue la permutation *identité*, et est assimilable à un arc (*cf.* remarque 3.40).

Remarque 3.48. Dans l'exemple 3.47, les séquences u et v doivent avoir le même débit ; dans le cas contraire une accumulation de jetons surviendrait dans l'une ou l'autre des places. Pour s'en convaincre, il suffit d'abstraire le trieur en graphe SDF et résoudre ses équations de balance (*cf.* sous-section 3.1.3).

Exemple 3.49. Différentes topologies de trieurs permettent d'effectuer une même permutation. Par exemple, considérons la permutation $\pi = [3,2,1]$. Nous proposons en FIG. 3.21 trois topologies différentes pour la réaliser : trieur parallèle (a), trieur série (b), et trieur itératif (c).

Trieur parallèle

La question des permutations a déjà été partiellement traité par la communauté étudiant la théorie des graphes, dans le cas d'un *trieur parallèle* ; des résultats généraux en sont résumés par Golumbic [101]. En transposant le principe dans notre modèle, nous en donnons la définition suivante.

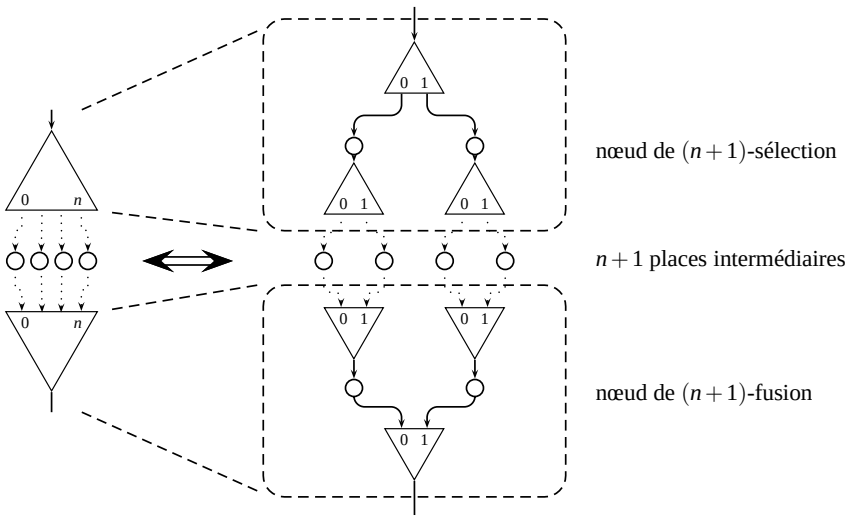


FIG. 3.22: Exemple de trieur parallèle.

Définition 3.50 (Trieur parallèle)**.** *Un trieur parallèle est un sous-KRG dédié à la permutation de jetons, dont les nœuds sont soit des sélections, soit des fusions, et connectés sur le modèle suivant :*

- *Les nœuds de sélection forment un arbre binaire, regroupés en un nœud de n-sélection.*
- *Les nœuds de fusion forment également un arbre binaire, regroupés en un arbre de n-fusion.*
- *Les deux arbres sont symétriques, et chaque feuille d'un arbre est reliée à son symétrique dans l'autre arbre, de telle façon que la sortie gauche (resp. droite) d'une sélection est connectée à l'entrée gauche (resp. droite) d'une fusion.*

Plus simplement, un trieur est constitué de files en parallèle, comme indiqué en FIG. 3.20(b). Un flot entrant dans le trieur, issu du sommet source, est distribué sur les files, via un arbre de sélection. L'arbre de fusion permet de regrouper les jetons issus des différentes files. La réalisation d'une permutation π repose alors sur le choix de la file où insérer chaque nouveau jeton, ainsi qu'à l'ordre dans lequel le sommet puits les consomme. Un exemple de trieur parallèle est donné en FIG. 3.22.

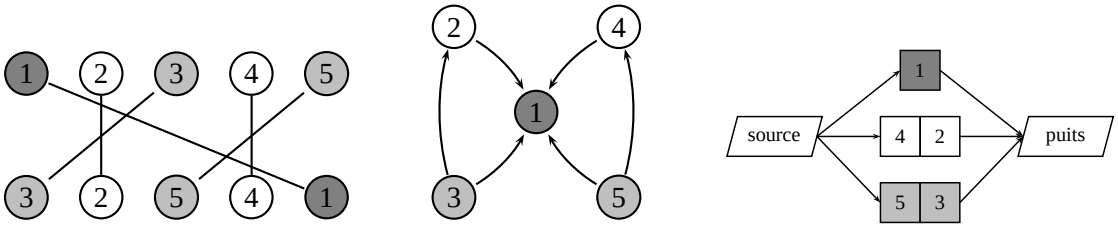
Comme montré par Golumbic [101], ce choix du placement des jetons revient à effectuer une coloration canonique du graphe de permutation $G[\pi]$. Il prouve la proposition suivante :

Proposition 3.51. *Soit π une permutation. Les nombres suivants sont égaux :*

- *le nombre chromatique χ de $G[\pi]$,*
- *le nombre minimum de files (ou chemins) nécessaires à la permutation π,*
- *la longueur de la plus longue sous-séquence décroissante de π.*

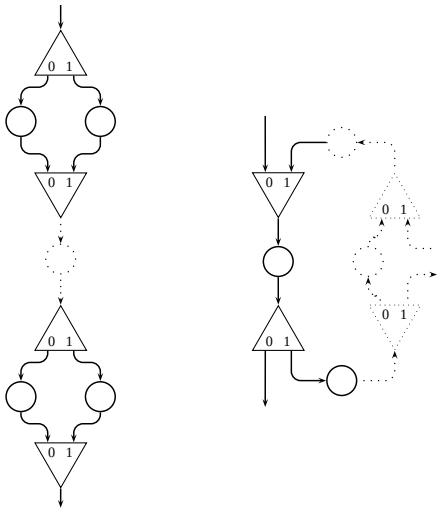
Comme nous l'avons déjà mentionné, deux jetons permutés ne peuvent être placés dans la même file : ils doivent emprunter des chemins différents. Par analogie, deux sommets adjacents du graphe de permutation ne peuvent être colorés de la même couleur.

Exemple 3.52. Considérons la permutation $\pi = [3,2,5,4,1]$ de la FIG. 3.23(a). Son graphe de permutation est donné en FIG. 3.23(b). La proposition 3.51 nous donne le nombre minimum de chemins nécessaire à la permutation : en l'occurrence, le graphe de permutation est 3-chromatique, donc trois chemins sont nécessaires et suffisants pour le trieur. De plus, la coloration du graphe de permutation nous donne directement l'affectation des jetons pour chaque file du trieur (cf. FIG. 3.23(c)).



(a) Diagramme de correspondance. (b) Graphe de permutation $G[\pi]$. (c) Permutation de jetons par un réseau de FIFOs en parallèle.

FIG. 3.23: Exemple de permutation $\pi = [3,2,5,4,1]$.



(a) Trieur série. (b) Trieur itératif.

FIG. 3.24: Autres topologies de trieurs.

Pour une χ-coloration minimale du graphe de permutation, on en déduit qu'un nœud de χ-sélection et un nœud de χ-fusion seront nécessaires et suffisants à un trieur parallèle minimal, soit un total de $\chi - 1$ nœuds de sélection, et autant de nœuds de fusion.

Proposition 3.53 (Routage du trieur parallèle)**.** *Soit $\pi = [\pi_1, \pi_2, ..., \pi_n]$ une permutation du n-uplet $[1,2,...,n]$. Le graphe de permutation $G[\pi]$ est X-chromatique. On note $\chi(i)$ l'indice de la couleur, compris entre 0 et $X-1$, associé au nœud i de $G[\pi]$ après coloration minimale. Soit s et f, respectivement, le nœud de X-sélection et le nœud de X-fusion d'un trieur minimal pour π. Les routages entiers de s et f, respectivement $R(s)$ et $R(f)$, sont tels que :*

$$R(s) = [\chi(1), \chi(2), ...\chi(n)] \quad (3.21)$$

$$R(f) = [\chi(\pi_1), \chi(\pi_2), ...\chi(\pi_n)] \quad (3.22)$$

Trieurs série et itératif

Nous ne mentionnerons que deux autres topologies de trieurs, sans les étudier plus en détails : le trieur série, et le trieur itératif. Ils sont sont représentés en FIG. 3.24.

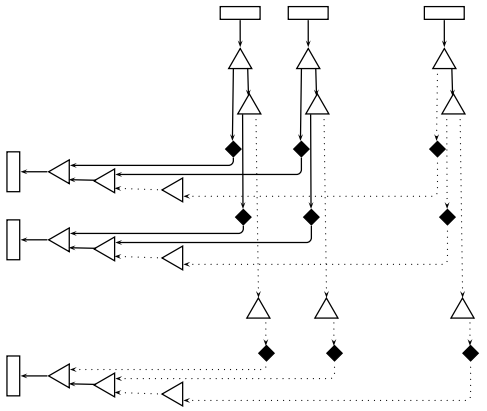


FIG. 3.25: Aspect général de la forme normale. Les nœuds de calcul sont représentés à la fois à gauche et en haut, et les places ne sont pas dessinées, afin de simplifier la figure. Les arbres de fusion et de sélection représentent le réseau d'interconnexion normalisé. Les carrés noirs représentent les trieurs.

Le mélangeur série, tout d'abord, repose sur le paradigme *diviser pour régner* (*cf.* FIG. 3.24(a)). On peut imaginer, pour ce type de trieur, de retenir en amont certains jetons pour se faire dépasser par d'autres, et ainsi échelonner les permutations tout au long du chemin. L'avantage d'une telle topologie est que seuls $\lceil \log_2 n \rceil$ nœuds de sélection, et autant de fusion, sont nécessaires pour disposer de n chemins dans le trieur. En revanche, il est beaucoup plus difficile de trouver un routage permettant de minimiser sa profondeur et sa capacité (*cf.* sous-section 3.5.3).

Le mélangeur circulaire permet quant à lui d'effectuer n'importe quelle permutation avec seulement un nœud de sélection et un nœud de fusion (*cf.* FIG. 3.24(b)). En contrepartie, les capacités des places et le temps nécessaire à effectuer une permutation augmentent de façon conséquente. Par exemple, nous souhaitons effectuer la permutation $\pi = [4,3,2,1]$. Les jetons «1», «2» et «3» doivent entrer dans la boucle afin de laisser passer «4». Puis les jetons «1» et «2» font un autre tour de boucle pour que «3» puisse sortir, et ainsi de suite. Ainsi, une inversion de n jetons demande $\sum_{i=1}^{n} 2i = (n+1)^2$ tirages de nœuds de routage dans le cas d'un tri circulaire, alors qu'un tri parallèle n'en coûte que $2n\log_2 n$. Un compromis consiste à composer plusieurs trieurs circulaires entre eux, ou y insérer des trieurs série : on y augmente le parallélisme du tri, et le cas dégénéré tend vers une forme de trieur série.

3.3.4 Expansion de l'interconnexion

Nous allons maintenant définir une normalisation de l'interconnexion du KRG, s'appuyant sur l'ensemble des points abordés dans cette section : transformations de la topologie du graphe, en particulier des interconnexions, et gestion des permutations de jetons.

Par soucis de simplicité, nous considérons tout d'abord le cas sans copies de jetons ; l'aspect général de cette forme normale est représenté en FIG. 3.25. Elle consiste à éclater chaque sortie des nœuds de calcul par un arbre de sélection, comme un peigne. Chaque branche est destinée à connecter le producteur à un et un seul consommateur. De façon symétrique, chaque entrée des nœuds de calcul est connectée à un arbre de fusion, regroupant ainsi les jetons issus des différents producteurs. Au milieu, les liens sont totalement éclatés, un pour chaque couple producteur/consommateur, à la façon d'un *crossbar*. Ici, des trieurs permuttent les jetons si nécessaire.

Théorème 3.54 (Complétude des transformations)**.** *Il existe une expansion en forme normale des KRG sans copie, pour laquelle les transformations* PlaceExp, PermFusion *et* PermSelect *sont complètes.*

Dans le cas d'un KRG avec copies de jetons, la forme normale est globalement la même. Les nœuds de sélection de la FIG. 3.25 sont pourtant remplacés par des sous-graphes plus complexes, à une entrée et deux sorties, contenant des boucles et des nœuds de copie : ces sous-graphes doivent être en mesure de produire des jetons sur chacune de leurs sorties, éventuellement dupliqués plusieurs fois. Une telle transformation demande à utiliser la règle de factorisation *FactPlace* (*cf.* FIG. 3.13(b)) qui, comme nous l'avons vu, n'est pas toujours possible. Nous conjecturons qu'elle est rendue possible dans ce cas particulier, et que le théorème de complétude peut être étendu au cas général.

Conjecture 3.55 (Complétude des transformations)**.** *Il existe une expansion en forme normale des KRG, pour laquelle les transformations* PlaceExp, PlaceFact, PermFusion *et* PermSelect *sont complètes.*

3.4 Factorisation et expansion des calculs

Dans la section précédente, nous avons défini un ensemble d'identités remarquables et de transformations menant à une expansion de l'interconnexion. Dans cette section, nous discutons informellement de transformations portant, quant à elles, sur les nœuds de calcul. Elles visent à scinder les flots de jetons entre différents nœuds de calcul pour, si possible, augmenter le parallélisme, ou au contraire, à entrelacer les flots pour séquentialiser les calculs.

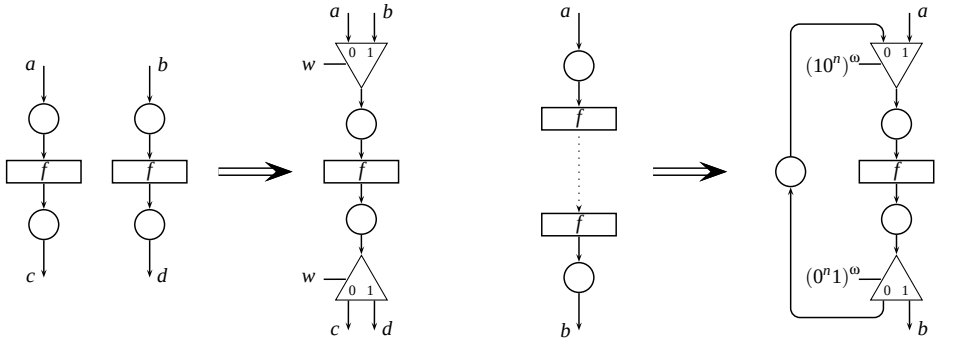
Nous ne présentons que des exemples de transformations possibles, afin de donner des indices intuitifs de l'expressivité du modèle. Leur formalisation n'est qu'un simple exercice ; la préservation des flots est prouvée de la même façon que pour les propositions 3.35 à 3.39.

3.4.1 Factorisation d'un calcul

Plusieurs nœuds de calculs peuvent être regroupés en un seul, ce qui revient à *factoriser* du code. Les flots sur chacune de leurs entrées sont entrelacés par des arbres de n-fusion, puis séparés en sortie par des arbres de n-sélection, n étant le nombre de nœuds factorisés. Les séquences de routage des nœuds de n-fusion et de n-sélection sont toutes égales, afin de préserver les flots. Cette transformation est semblable à la factorisation d'une place, évoquée dans la section précédente. La différence essentielle entre les factorisation de places et de nœuds est que dans le cas présent, nous interdisons les «chemins croisés» : le cas de l'exemple 3.36, qui introduisait un blocage, est ici impossible. Notons cependant que la fusion de nœuds n'est possible qu'à condition que les nœuds soient fonctionnellement égaux, donc qu'ils aient les mêmes labels.

Exemple 3.56. Une forme générale de factorisation est représentée en FIG. 3.26(a). Si les flots a et b sont fusionné selon un routage w, arbitraire, alors la séquence de routage de la sélection est également w. Tous les jetons venant de a sont alors dirigés vers c, et de même entre b et d.

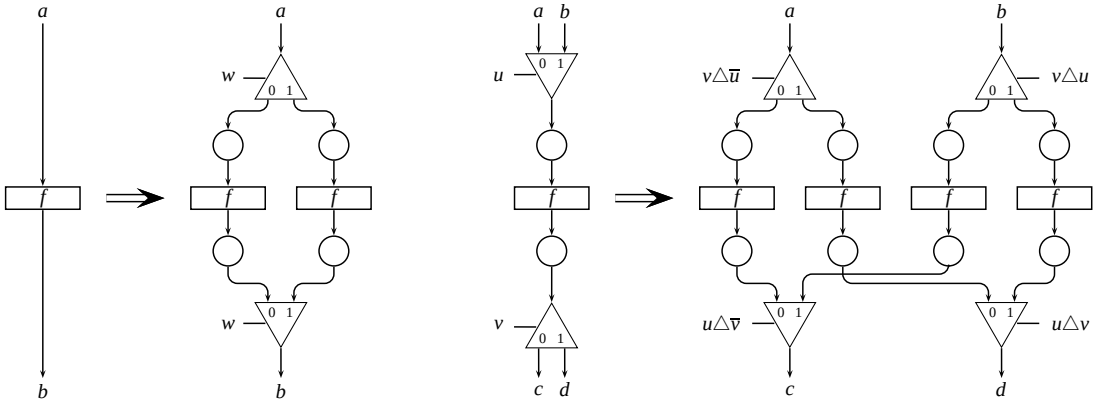
Remarque 3.57. Un cas particulier de la factorisation consiste à l'appliquer sur des nœuds en séquence, comme en FIG. 3.26(b). En insérant des nœuds de n-fusion et de n-sélection, puis en simplifiant les chemins, nous créons des boucles entre nœuds de fusion et de sélection. La fusion sert à l'initialisation ou au passage à l'itération suivante, tandis que la sélection représente la condition



(a) Factorisation de blocs en parallèle.

(b) Factorisation de blocs en séquence, introduction d'une boucle.

FIG. 3.26: Factorisation de nœuds de calcul.



(a) (b)

FIG. 3.27: Expansion d'un nœud de calcul.

d'arrêt : sortie de boucle, sinon redirection vers l'entrée, à la façon d'un «`goto label`». Pour une séquence de n nœuds de calcul, les routages des fusions et des sélections sont symétriques : les premiers valent $\left(1.0^{n-1}\right)^\omega$, et les seconds $\left(0^{n-1}.1\right)^\omega$.

3.4.2 Expansion d'un calcul

À l'inverse du point précédent, les flots peuvent être scindés et traités par des nœuds de calcul concurrents. Cette forme expansée permet, lorsque cela est possible, d'exploiter plus de parallélisme dans l'application.

Remarque 3.58. Notons que le fait de dupliquer un nœud de calcul ne permet pas *systématiquement* d'introduire du parallélisme. Le fait que deux calculs puissent être effectués en parallèle est une propriété *sémantique*, tandis que la duplication de nœuds est une transformation *syntaxique*. Cette transformation présente un intérêt au cas par cas, qui demande une analyse des dépendances de données en amont.

Exemple 3.59. Deux exemples d'expansion de nœud de calcul sont présentées en FIG. 3.27. Le cas (a) est corollaire à la remarque 3.40. Le cas (b), quant à lui, est semblable à l'expansion d'une place

ExpPlace (*cf.* sous-section 3.3.1).

3.5 Graphes à routage *k*-périodique synchrones

Les réseaux de processus flot de données «purs», tels que les graphes marqués ou SDF, ont été un terrain d'études intensives sur l'ordonnancement, afin de situer les calculs dans un temps unique. Des séquences binaires, faisant office de condition d'activation, indiquent l'activité d'un nœud à chaque instant. Un nœud peut alors s'exécuter pour l'unique raison que la présence de jetons sur ses entrées à cet instant est prouvée statiquement, lors du calcul de cet ordonnancement. En particulier, nous nous intéressons à des ordonnancements périodiques (ou ultimement périodiques). Les capacités des places sont alors optimisées en conséquence.

Ainsi, nous raffinons le modèle KRG pour intégrer des notions de temps et d'espace : nous introduisons des contraintes de latence et de capacité, afin de calculer les tailles des mémoires et les horloges d'activation des différents composants. Nous appelons ces extensions des graphes à routage *k*-périodique synchrones *(Synchronous K-periodically Routed Graph* – SKRG).

3.5.1 Définition

Définition 3.60 (Graphe à routage *k*-périodique synchrone)**.** *Un* graphe à routage *k*-périodique synchrone *est un octuplé* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R, K, L, T \rangle$*, tel que :*

- $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R \rangle$ *est un graphe à routage k-périodique*
- *K est la fonction attribuant une capacité à chaque place, nombre maximum de jetons qu'elle peut contenir simultanément*
$$K : \mathcal{P} \to \mathbb{N} \cup \{+\infty\} \tag{3.23}$$
- *L est la fonction attribuant des latences aux nœuds et places, nombre d'instants logiques nécessaires à leur franchissement par un jeton.*
$$L : \begin{cases} \mathcal{N}_\lambda \cup \mathcal{P} \to \mathbb{N} \\ \mathcal{N}_c \cup \mathcal{N}_s \cup \mathcal{N}_f \to \{0\} \end{cases} \tag{3.24}$$
- *T est la fonction d'ordonnancement, associant à chaque nœud et place une séquence binaire ultimement périodique.*
$$T : \mathcal{N} \cup \mathcal{P} \to \mathbb{P} \tag{3.25}$$

Par rapport à la définition 3.1 des KRG, la définition 3.60 des SKRG introduit trois nouveaux concepts :

Latences. Les latences de calcul et de communication sont inspirées des graphes marqués temporisés (*cf.* définition 2.12) de façon à conduire des analyses temporelles. Ces latences sont entières et quelconques ; or dans le cas synchrone, nous considérons les productions et consommations de jetons à chaque instant. Nous pouvons alors transformer le graphe par expansion, de façon similaire à celle des réseaux de Petri temporisés [200] ou des graphes marqués [33, 39] : pour chaque nœud de latence non-nulle (resp. place de latence non-unitaire) il (elle) est découpé(e) et remplacé(e) en introduisant des nœuds intermédiaires de latence nulle et des places de latence unitaire. Si un nœud de calcul de latence non-nulle modélise une fonction ne pouvant être exécutée en pipeline, on ajoute une place entre le dernier et le premier étage ; la non-réentrance est alors assimilée à une dépendance de données [14] (*cf.* FIG. 3.28).

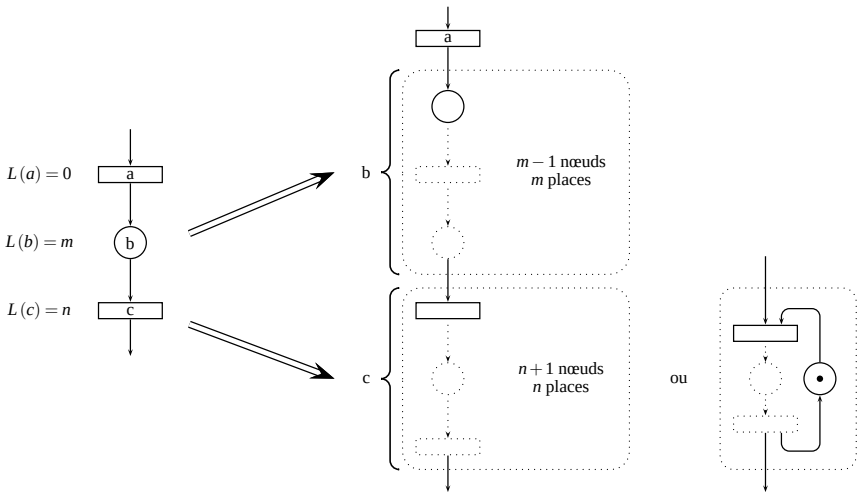


FIG. 3.28: Expansion des latences, selon qu'un nœud de calcul et exécuté en pipeline *(gauche)* ou non *(droite)*.

Capacités. La capacité d'une place est le nombre maximum de jetons qu'une place peut stocker à un instant donné. Une place de capacité infinie signifie que le concepteur n'a pas borné la mémoire considérée, mais n'a pas pour autant de réalité physique. Nous devons alors borner la capacité de la place, sans pour autant réduire les débits des nœuds (*cf.* définition 3.64 et sous-section 2.2).

Ordonnancement. Contrairement aux KRG, nous introduisons dans les SKRG une notion de temps, en suivant l'approche originelle menée avec les graphes marqués ou SDF. Selon le paradigme synchrone, ce temps est discrétisé en instants logiques. L'exécution d'un SKRG est une succession de réactions de ses nœuds et places au cours du temps. Les horloges d'activation des nœuds et places forment un système polychrone, dont les relations entre horloges sont définies selon des règles d'activation et de tirage. Cependant, nous souhaitons raffiner ces relations, et définissons un SKRG comme un système synchrone : une *horloge maîtresse* échantillonne l'ensemble des instants logiques ; toute autre horloge en est une sous-horloge. Ainsi, les horloges d'activation des nœuds et places sont représentées par des séquences binaires, ou *ordonnancements*. Si la $i^{\text{ème}}$ lettre d'une séquence vaut «1», alors le signal de l'horloge correspondante est présent au $i^{\text{ème}}$ instant logique : le nœud est tiré, ou un jeton arrive dans la place. Sinon, le signal de l'horloge est absent, ce qui équivaut à une attente.

Remarque 3.61. Les ordonnancements sont ultimement périodiques. C'est précisément pour cette raison que nous avions défini les séquences de routage comme ultimement périodiques. En effet, il existe un lien fort entre ordonnancement et routage : il n'est possible de calculer un ordonnancement périodique (ou *k*-périodique) qu'à la condition que les séquences de routage le soient aussi.

Ce lien entre routage est ordonnancement est un point important. Le fait de transformer l'interconnexion peut, quand c'est possible, permettre de «*balancer*»[3] les ordonnancements.

Par la suite, nous supposons que tout circuit contient au moins une place ou un nœud de latence non-nulle ; nous excluons la possibilité qu'un SKRG puisse contenir un circuit combinatoire.

[3]Ici, *balancer* est utilisé au sens des séquences binaires (*cf.* Millo [166]), et non des équations du même nom en SDF.

3.5.2 Ordonnancement

Dans la sous-section 3.1.3, nous avons vu que le nombre de tirage de chaque nœud au cours d'une période peut être calculé par abstraction en SDF. Maintenant, nous recherchons un ordonnancement à grain plus fin, pour positionner les exécutions des nœuds les unes par rapport aux autres, au sein même d'une période.

Dans cette sous-section, nous abordons le problème du calcul d'un ordonnancement pour un graphe donné, ou de la décision de sa consistance, c'est-à-dire prouver qu'un ordonnancement permet l'exécution d'un SKRG tout en respectant les règles d'activation et de tirage des nœuds.

Règle 3.62 (Activation)**.**
- *Un jeton dans une place p est prêt à être consommé à un instant donné s'il a été produit dans p au moins $L(p)$ instants auparavant. Les jetons initiaux sont consommables dès le premier instant d'exécution.*
- *Un nœud de calcul, de copie ou de sélection est activé si et seulement si chaque place en entrée contient au moins un jeton prêt à être consommé.*
- *Un nœud de fusion n est activé si et seulement si sa place en entrée, indiquée par la première lettre de la séquence de routage, contient au moins un jeton prêt à être consommé.*

Règle 3.63 (Tirage)**.**
- *Seul un nœud activé peut être tiré.*
- *Le tirage d'un nœud calcul n consomme un jeton dans chaque place en entrée. Un jeton est produit dans chaque place en sortie $L(n)$ instants plus tard.*
- *Le tirage d'un nœud de copie consomme un jeton dans la place en entrée. Un jeton est produit simultanément dans chaque place en sortie.*
- *Le tirage d'un nœud de fusion consomme un jeton dans la place indiquée par la séquence de routage. Un jeton est produit simultanément dans la place en sortie. La première lettre de la séquence de routage est consommée.*
- *Le tirage d'un nœud de sélection consomme un jeton dans sa place en entrée. Un jeton est produit simultanément dans la place en sortie indiquée par la première lettre de la séquence de routage. Celle-ci est consommée.*

Définition 3.64 (Débit)**.** *Le* débit d'un nœud *est le débit de la partie périodique de son ordonnancement. Le* débit d'un SKRG *est la liste des débits de ses entrées (sources) et sorties (puits).*

Le débit d'un graphe acyclique est égal à 1. Contrairement aux graphes dépourvus de contrôle, le débit d'un KRG n'est pas nécessairement égal au débit de son cycle (ou chemin) le plus lent.

Exemple 3.65. Considérons le SKRG de la FIG. 3.29 : le SKRG a un débit de $1/2$ (débit en sortie de la copie), alors qu'il contient deux nœuds de débit $1/4$.

Caractère borné

Comme nous l'avons vu, un KRG et un SKRG sont bornés si et seulement si les productions des jetons compensent les consommations. Ceci dépend donc des rythmes auxquels sont tirés les nœuds. Dans le cas d'un KRG, asynchrone, ce problème est donc résolu par abstraction du graphe en SDF et résolution des équations de balance (*cf.* règle 3.7) : nous étudions les productions et consommations dans chaque place au cours d'une période du graphe, et recherchons les rythmes de tirages des nœuds de façon à équilibrer le tout.

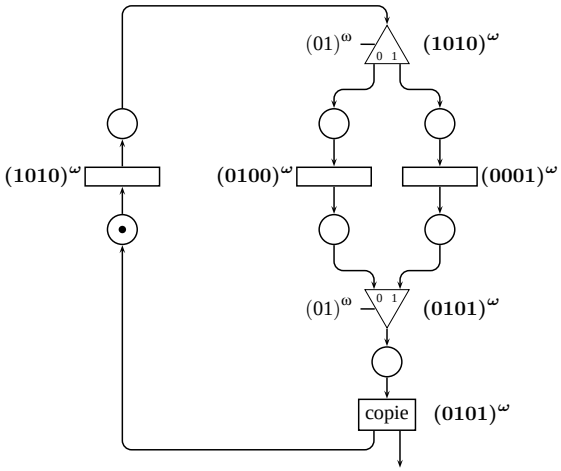


FIG. 3.29: Ordonnancement (en gras) d'un KRG de débit $1/2$.

Dans le cas synchrone d'un SKRG, nous transposons ce raisonnement en un calcul d'horloges : nous posons une contrainte sur l'ensemble des ordonnancements du SKRG, afin de ne considérer que ceux correspondant à un comportement balancé. Ceci est étroitement liée à la notion de *synchronisabilité* des ordonnancements (*cf.* Cohen *et al.* [63] et définition A.11). Plus précisément, la synchronisabilité est une spécialisation du critère de Lee-Messerschmitt aux horloges. Chercher une hyper-période sur laquelle les activités des producteurs et des consommateurs sont à l'équilibre revient donc à leurs recherches des ordonnancements tels que leurs débits soient égaux. Ce qui est équivalent à synchroniser leurs horloges (proposition A.13).

Absence de blocage

L'absence de blocage, et plus particulièrement la quasi-vivacité, sont quant à elles des propriétés qui dépendent du marquage initial. Nous avons étudié ce problème dans le cas asynchrone ; pour un SKRG, nous cherchons maintenant un ordonnancement, s'il existe, respectant les relations sur ses flots (*cf.* sous-section 3.1.2). Nous en déduisons des contraintes sur les ordonnancements des nœuds et places.

Par abus de notation, nous désignons par $\bullet_0 n$ ou $\bullet_1 n$ les places connectées respectivement aux entrées «0» et «1» du nœud de fusion n, et $n'\bullet_0$ et $n'\bullet_1$ les places connectées respectivement aux sorties «0» et «1» du nœud de sélection n'.

Lemme 3.66 (Contraintes des calculs et copies)**.** *Pour tout nœud de calcul ou de copie n, à tout instant i, les règles de tirages imposent les contraintes suivantes sur les ordonnancements :*

$$\forall p \in \bullet n, \quad \mathrm{idx}_1(T(p), i) + L(p) \leq \mathrm{idx}_1(T(n), i + M(p)) \tag{3.26}$$

$$\forall p \in n\bullet, \quad \mathrm{idx}_1(T(n), i) + L(n) = \mathrm{idx}_1(T(p), i) \tag{3.27}$$

Les équations du lemme 3.66 peuvent aussi s'exprimer sous la forme de contraintes d'horloge en CCSL, telles que :

$$(3.26) \quad \Leftrightarrow \quad \forall p \in \bullet n, \ (T(p)(L(p)) \rightsquigarrow \tau)^\omega \preccurlyeq T(n) \blacktriangledown \left(0^{M(p)}.1^\omega\right) \tag{3.28}$$

$$(3.27) \quad \Leftrightarrow \quad \forall p \in n\bullet, \ (T(n)(L(n)) \rightsquigarrow \tau)^\omega \equiv T(p) \tag{3.29}$$

où τ est l'horloge absolue, présente à chaque instant logique.

Lemme 3.67 (Contraintes des fusions)**.** *Pour tout nœud de fusion n, à tout instant i, les règles de tirages imposent les contraintes suivantes sur les ordonnancements :*

$$\begin{aligned}
\mathrm{idx}_1\left(T\left(\bullet_0 n\right), i\right) + L\left(\bullet_0 n\right) &\leq \mathrm{idx}_1\left(T\left(n\right) \blacktriangledown \overline{R\left(n\right)}, i + M\left(\bullet_0 n\right)\right) && (3.30)\\
\mathrm{idx}_1\left(T\left(\bullet_1 n\right), i\right) + L\left(\bullet_1 n\right) &\leq \mathrm{idx}_1\left(T\left(n\right) \blacktriangledown R\left(n\right), i + M\left(\bullet_1 n\right)\right) && (3.31)\\
\mathrm{idx}_1\left(T\left(n\right), i\right) &= \mathrm{idx}_1\left(T\left(n\bullet\right), i\right) && (3.32)
\end{aligned}$$

Par le lemme 3.67, on obtient les contraintes suivantes sur les fusions, exprimées en CCSL :

$$\begin{aligned}
(3.30) &\Leftrightarrow \left(T\left(\bullet_0 n\right)\left(L\left(\bullet_0 n\right)\right) \rightsquigarrow \tau\right)^{\omega} \preccurlyeq \left(T\left(n\right) \blacktriangledown \overline{R\left(n\right)}\right) \blacktriangledown \left(0^{M\left(\bullet_0 n\right)}.1^{\omega}\right) && (3.33)\\
(3.31) &\Leftrightarrow \left(T\left(\bullet_1 n\right)\left(L\left(\bullet_1 n\right)\right) \rightsquigarrow \tau\right)^{\omega} \preccurlyeq \left(T\left(n\right) \blacktriangledown R\left(n\right)\right) \blacktriangledown \left(0^{M\left(\bullet_1 n\right)}.1^{\omega}\right) && (3.34)\\
(3.32) &\Leftrightarrow T\left(n\right) \equiv T\left(n\bullet\right) && (3.35)
\end{aligned}$$

Lemme 3.68 (Contraintes des sélections)**.** *Pour tout nœud de sélection n, à tout instant i, les règles d'activation et de tirage imposent les contraintes suivantes sur les ordonnancements :*

$$\begin{aligned}
\mathrm{idx}_1\left(T\left(\bullet n\right), i\right) + L\left(p\right) &\leq \mathrm{idx}_1\left(T\left(n\right), i + M\left(\bullet n\right)\right) && (3.36)\\
T\left(n\bullet_0\right) &= T\left(n\right) \blacktriangledown \overline{R\left(n\right)} && (3.37)\\
T\left(n\bullet_1\right) &= T\left(n\right) \blacktriangledown R\left(n\right) && (3.38)
\end{aligned}$$

La règle 3.68 nous donne les contraintes suivantes sur les sélections, exprimées en CCSL :

$$\begin{aligned}
(3.36) &\Leftrightarrow \left(T\left(\bullet n\right)\left(L\left(\bullet n\right)\right) \rightsquigarrow \tau\right)^{\omega} \preccurlyeq T\left(n\right) \blacktriangledown \left(0^{M\left(\bullet n\right)}.1^{\omega}\right) && (3.39)\\
(3.37) &\Leftrightarrow T\left(n\bullet_0\right) \equiv T\left(n\right) \blacktriangledown \overline{R\left(n\right)} && (3.40)\\
(3.38) &\Leftrightarrow T\left(n\bullet_1\right) \equiv T\left(n\right) \blacktriangledown R\left(n\right) && (3.41)
\end{aligned}$$

Du fait qu'un SKRG est confluent, n'importe quel ordonnancement, respectant ces containtes sur les consommations et productions de jetons, préservera l'absence de blocage et la quasi-vivacité.

Équations de balance augmentées

Kerihuel *et al.* étendent les équations de balance de SDF en y intégrant une notion de temps [129]. L'idée est que non seulement les productions et consommations doivent se compenser, comme mentionné ci-dessus, mais qu'en plus, chaque composant d'un système VLSI (ici, les nœuds du graphe) «consomme» les signaux sur ses entrées dans l'instant où ils sont émis (*i.e.* que les jetons sont produits). Cette contrainte est trop forte pour notre modèle, les jetons pouvant être conservés plusieurs instants dans une place avant d'être consommés. En revanche, le principe peut être adapté aux équations de balances des SKRG pour y introduire une notion de causalité : un jeton doit être produit avant d'être consommé.

Pour résumer, un ordonnancement est consistant pour un SKRG si les productions et consommations des nœuds sont balancées, et s'il respecte les contraintes imposées par les relations sur les flots de jetons. D'où le théorème suivant.

Théorème 3.69 (Équations de balance augmentées)**.** *Une fonction d'ordonnancement T est consistante pour un KRG donné si et seulement si :*

1. *Les débits des producteurs et des consommateurs sont égaux deux à deux :*

$$\forall p \in \mathcal{P},\ \ T(\bullet p) \bowtie T(p\bullet) \tag{3.42}$$

2. *T respecte les contraintes sur les ordonnancements des lemmes 3.66, 3.67 et 3.68.*

Pour un SKRG donné, on peut lui calculer un ordonnancement valide soit par simulation, soit par résolution des systèmes de contraintes à l'aide d'un solveur adéquat [10].

3.5.3 Capacités des places et trieurs

Maintenant que nous savons calculer les ordonnancements des nœuds, nous pouvons en déduire les capacités minimales des places permettant de ne pas ralentir les débits.

Capacités des places

Proposition 3.70 (Capacité d'une place)**.** *Soit p une place d'un SKRG borné, entre un producteur n_1 et un consommateur n_2. Sa capacité minimale $K_{min}(p)$ est donnée par la formule suivante :*

$$K_{min}(p) = \max_t(\alpha - \beta) + M(p) \tag{3.43}$$

où α et β sont tels que :

$$\alpha = \begin{cases} |\text{prf}(R(n_1), |\text{prf}(T(n_1), t)|_1)|_{b_1} & \text{si } n_1 \text{ est un nœud de sélection} \\ |\text{prf}(T(n_1), t)|_1 & \text{sinon} \end{cases} \tag{3.44}$$

$$\beta = \begin{cases} |\text{prf}(R(n_2), |\text{prf}(T(n_2), t)|_1)|_{b_2} & \text{si } n_2 \text{ est un nœud de fusion} \\ |\text{prf}(T(n_2), t)|_1 & \text{sinon} \end{cases} \tag{3.45}$$

avec b_1 et b_2 les labels de l'entrée et de la sortie à laquelle est connectée p, dans le cas où le producteur ou le consommateur sont des nœuds de routage.

Capacités des trieurs

Rappelons tout d'abord que le tirage d'un nœud de *n*-fusion ou de *n*-sélection est atomique (*cf.* sous-section 3.3.2) : tout jeton consommé est aussitôt produit en sortie. Ceci nous est nécessaire pour modéliser des multiplexeurs ou démultiplexeurs, dépourvus de mémoires internes. Si l'on considère l'arbre équivalent, les places s'y comportent comme de simples fils. Ainsi, pour un trieur constitué d'une *n*-sélection, d'une *n*-fusion, et de places intermédiaires, ses seules capacités de stockage se situent au niveau desdites places. Cette hypothèse permet de faire coïncider la sémantique d'un trieur avec sa représentation intuitive en FIG. 3.20(b).

Définition 3.71 (Capacité d'un trieur)**.** *On appelle* capacité *d'un trieur la somme des capacités de ses places intermédiaires ; c'est donc le nombre maximum de jetons pouvant être stockés simultanément dans le trieur.*

Un jeton est stocké dans le trieur dans trois cas distincts :
– il est le prochain jeton à sortir du trieur,
– au moins l'un de ses successeurs doit le dépasser,

– au moins l'un de ses prédécesseurs doit sortir avant lui, lui-même stocké pour l'une des deux raisons précédentes.

Notons qu'un jeton peut traverser le trieur combinatoirement. Toutes les places intermédiaires peuvent avoir une capacité non nulle ; inversement, au plus une place intermédiaire de capacité nulle est nécessaire. Le placement des mémoires d'un trieur de capacité n sur ses p places intermédiaires est donc une partition de n en $p-1$ ou p parties. Ce partitionnement dépend de l'allocation des jetons sur les chemins du trieur, donc de la coloration du graphe de permutation (*cf.* sous-section 3.3.3). Le problème d'un algorithme de coloration *first fit*[4] est qu'il ne permet pas, dans le cas général, d'aboutir à un placement minimisant la capacité du trieur.

Exemple 3.72. Considérons la permutation $\pi = [4,2,3,1,8,7,5,6]$. Elle est composée de deux permutation distinctes, $[4,2,3,1]$ d'une part, et $[8,7,5,6]$ d'autre part. Leurs graphes de permutations sont tous deux 3-chromatiques. Un algorithme *first fit* nous donne les affectations suivantes :

– Affectations pour la première permutation :
  File 1 : (1)
  File 2 : $(2,3)$
  File 3 : (4)
– Affectations pour la seconde permutation :
  File 1 : $(5,6)$
  File 2 : (7)
  File 3 : (8)

Un minimum de deux places serait donc nécessaire sur chacune des deux premières files. Les jetons 4 et 8 peuvent franchir le trieur combinatoirement dans le meilleur cas ; une capacité totale de quatre unités est donc nécessaire. Il est pourtant évident qu'en faisant correspondre la file 2 de la première permutation avec la file 1 de la seconde permutation, nous ferions chuter la capacité minimale à seulement trois unités.

Sur cet exemple, une première idée serait de colorer indépendamment les permutations distinctes, puis de trier les files par capacité croissante, à la façon d'un diagramme de Ferrers[5] ; en faisant correspondre les files, nous en déduisons les capacités minimales de chacune, et donc la capacité minimale du trieur. Pourtant, cette solution est toujours insuffisante.

Exemple 3.73. Prenons maintenant la permutation $\pi = [5,3,7,4,2,1,6]$; son graphe est 4-chromatique. Par un algorithme *first fit*, on obtient le placement suivant dans le trieur :

File 1 : $(1,6)$
File 2 : (2)
File 3 : $(3,4)$
File 4 : $(5,7)$

À supposer que les jetons 5 et 7 franchissent le trieur combinatoirement, cinq unités de stockage sont donc nécessaires pour effectuer la permutation avec cette allocation. Soit $T(n)$ l'ordonnancement du nœud de 4-sélection ; le jeton 5 franchit donc le mélangeur à l'instant $\mathrm{idx}_1(T(n),5)$. Supposons que le jeton 3 sort à l'instant $\mathrm{idx}_1(T(n),5)+1 \leq \mathrm{idx}_1(T(n),6)$. Ainsi, le jeton 6 entre dans le trieur alors que le jeton 3 sort dans le même instant, ou est déjà sorti. Si le jeton 6 avait été dirigé vers le troisième canal, nous aurions réduit la capacité minimale du trieur à quatre unités.

[4] La première couleur satisfaisant les contraintes de coloration est associée à chaque sommet du graphe. C'est, par exemple, le cas de l'algorithme de coloration de Supowit [101, 222].

[5] *cf.* Comtet [67] pour une définition et des exemples.

Ces exemples mettent en avant deux contraintes : *(i)* le placement des jetons les uns par rapport aux autres peut influer sur le résultat final, qu'ils soient permutés ou non, et par conséquent, *(ii)* il est impératif de tenir compte de la durée de vie de chaque jeton dans le trieur. Il apparaît alors que notre problème se ramène à celui plus général de l'allocation de registres [82]. Ce problème, bien connu pour son caractère NP-complet, est souvent abordé par la coloration du graphe d'interférence des données. Un graphe d'interférence est un graphe d'intervalle, dont les intervalles sont délimités par la date d'allocation d'une variable et sa date de libération.

Dans le contexte de l'allocation de registres d'un CPU, ces dates sont obtenues par transformation et analyse du programme sous forme SSA, voire SSI [43]. Dans notre cas, appelons n_1 le nœud de n-sélection et n_2 le nœud de n-fusion. Le $i^{\text{ème}}$ jeton entre dans le trieur à l'instant $\alpha_i = \text{idx}_1\left(T\left(n_1\right), i\right)$, et l'on peut récursivement calculer sa date de sortie au plus tôt δ_i par :

$$\delta_i = \max\left(\text{idx}_1\left(T\left(n_1\right), i\right), \delta_{\pi_{\left(\pi_i^{-1}-1\right)}} + 1\right) \tag{3.46}$$

La méthode proposée par Grund-Hack [112] est une solution optimale par programmation linéaire en nombres entiers pour optimiser la réutilisation de registres. Elle se base sur celle d'Appel-George [11], tout en diminuant significativement le temps de calcul. Le principe est le suivant : au graphe d'interférence est ajoutée une nouvelle famille d'arcs, les arcs d'affinité. Partant d'une coloration initiale, on cherche à maximiser le nombre de couples de nœuds de même couleur reliés par des arcs d'affinité, tout en s'assurant que tout arc d'interférence relie deux nœuds de couleurs différentes.

Notre problème est cependant quelque peu différent de celui de la simple allocation de registres d'un CPU. Dans leur cas, le nombre de registres y est connu ; l'enjeu est de trouver un agencement des données permettant la meilleure réutilisation possible. Dans notre cas, la capacité du mélangeur est *a priori* inconnue, et nous cherchons un routage minimisant le nombre nécessaire. Il nous faut donc se baser sur le principe des méthodes existantes, tout en adaptant leur formulation.

Nous nous proposons alors de formuler un programme quadratique en nombres entiers permettant de calculer la capacité minimale d'un mélangeur, inspiré de ceux de Grund-Hack et Appel-George. Soit $G_P\left[\pi\right] = (V_P, E_P)$ et $G_I\left[\pi\right] = (V_I, E_I)$, respectivement les graphes de permutation et d'interférence, tous deux non-orientés, d'une permutation π. On note X la coloration minimale de $G_P\left[\pi\right]$. On définit un ensemble de paramètres tels que :

- $C_i \overset{\text{def}}{=} 1$ si $\alpha_i = \delta_i$, 0 sinon.
- $I_{i,j} \overset{\text{def}}{=} 1$ si $(i, j) \in E_I$, 0 sinon.
- $P_{i,j} \overset{\text{def}}{=} 1$ si $(i, j) \in E_P$, 0 sinon.

Aussi, on définit les variables suivantes, sur lesquelles va travailler le solveur :

- $\chi_{i,k} = 1$ si le nœud i est de couleur k, 0 sinon.
- c_k est la capacité de la $k^{\text{ème}}$ place, associée à la $k^{\text{ème}}$ couleur.

Proposition 3.74 (Capacité minimale d'un trieur pour nombre de chemins minimal)**.** *Considérons un trieur de n jetons, de nombre de chemin minimal X. Sa capacité minimale est donnée par le programme linéaire en nombres entiers suivant :*

$$\text{minimiser } \mathit{capacité} = \sum_{k=1}^{X} c_k \tag{3.47}$$

avec :

$$\chi_{i,k} \in \{0,1\}, \quad c_k \in \mathbb{N} \qquad \forall i \in [\![1,n]\!], \forall k \in [\![1,X]\!] \tag{3.48}$$

$$\sum_{k=1}^{X} \chi_{i,k} = 1 \qquad \forall i \in [\![1,n]\!] \tag{3.49}$$

$$\left(\chi_{i,k} + \chi_{j,k}\right) P_{i,j} \leq 1 \qquad \forall i,j \in [\![1,n]\!], \forall k \in [\![1,X]\!] \tag{3.50}$$

$$\left(\left(\sum_{j=1}^{i-1} \chi_{j,k} I_{j,i}\right) + 1 - C_i\right) \chi_{i,k} \leq c_k \qquad \forall i \in [\![1,n]\!], \forall k \in [\![1,X]\!] \tag{3.51}$$

La proposition 3.74 nous donne, pour un nombre minimal de chemins, la capacité c_k de chaque place, et le routage $\chi_{i,k}$ adéquat. On note $\chi_i = k$ si et seulement si $\chi_{i,k} = 1$. On peut étendre ce programme pour déterminer si, pour un trieur donné, il est possible d'effectuer une certaine permutation π. Pour ce faire, nous ajoutons des contraintes de type $c_k \leq M_k$, où M_k est la capacité imposée à la place k. Si le système de contraintes n'a pas de solution, alors la permutation n'est pas consistante par rapport au trieur. Au contraire, si une solution existe, alors un routage χ_i est attribué à tout jeton i.

Une autre amélioration possible consiste à de ne pas se limiter au nombre de chemins minimum. Des nœuds de n-sélection et n-fusion correspondent respectivement à des multiplexeurs et des démultiplexeurs. Selon l'architecture sous-jacente, il peut être utile, par exemple, d'arrondir le nombre de chemins à la puissance de deux supérieure. Cela revient à remplacer le paramètre X de la proposition 3.74 par $X' = 2^{\lceil \log_2 X \rceil}$.

Proposition 3.75 (Temps de tri parallèle)**.** *Le temps nécessaire à effectuer la permutation est optimal avec un trieur parallèle.*

Deuxième partie

Du programme au modèle

Chapitre 4

Modèle polyédrique

Même si un jour la machine acquérait un certain esprit de géométrie, il lui manquerait encore l'esprit de finesse.

Général de corps d'armée Desfemmes, Réflexions sur la guerre électronique. *L'Armée*, décembre 1962.

Sommaire

Dans ce chapitre, nous décrivons le modèle polyédrique, aujourd'hui largement utilisé dans le domaine de la parallélisation automatique de programmes. C'est un modèle offrant un haut niveau d'abstraction à partir duquel, dans le chapitre suivant, nous construirons un KRG. L'expressivité des KRG nous permettra alors d'exploiter le parallélisme explicite calculé au niveau polyédrique. Aussi notre présentation est orientée vers les résultats du chapitre 5.

Le modèle polyédrique est né des travaux initiateurs de Karp *et al.* sur les systèmes d'équations récurrentes uniformes [71, 128]. Il offre une représentation algébrique et sémantique des relations entre données, au grain le plus fin : il manipule directement des *instances d'instructions*, ou *opérations*. La

puissance du modèle polyédrique repose sur sa capacité à modéliser des compositions de transformations linéaires, et sa flexibilité pour l'expression d'heuristiques d'optimisation : «*la puissance du modèle polyédrique est d'implanter des compositions de transformations complexes en une seule opération algébrique, et de modéliser ces transformations dans un espace d'optimisation convexe*» [22].

L'utilisation de la théorie des polyèdres s'est propagée dans de nombreuses communautés ; elle est aujourd'hui utilisée dans les domaines de la compilation [17, 72, 99], de l'allocation de mémoire [73, 111] ou encore de la vérification formelle [171]. En particulier, le modèle polyédrique est particulièrement adapté pour représenter des applications flots de données, avec un contrôle limité et répétitif. Ces applications sont principalement décrites de deux façons : soit sous la forme déclarative de *systèmes d'équations*, soit sous forme de *nids de boucles* dans un langage impératif.

Dans ce chapitre, nous présentons les travaux sur le modèle polyédrique, tel qu'il est utilisé dans le contexte de la compilation et la parallélisation automatique. Dans une première section, nous présentons les différentes approches menant au modèle polyédrique. Puis nous rappelons les définitions des différents concepts du modèle polyédrique. Dans la dernière section, nous abordons le problème de la recherche d'un ordonnancement au sein du modèle polyédrique.

4.1 Différentes approches

4.1.1 Systèmes d'équations récurrentes

Les premières sources d'inspiration du modèle polyédrique sont les travaux sur les systèmes d'équations récurrentes uniformes (SERU) [72, 128]. Ces travaux furent par la suite étendus aux systèmes d'équations récurrentes affines (SERA) [164].

Système d'équations récurrentes

De façon générale, un système d'équations récurrentes est un ensemble d'équations de la forme :

$$v_i(\vec{x}) = F_i(v_{j_1}(G_{i,j_1}(\vec{x})), \ldots, v_{j_n}(G_{i,j_n}(\vec{x}))) \tag{4.1}$$

où :

- les v_i sont des variables spatiales, fonctions d'un domaine de définition $\mathcal{D}_i$ de points de $\mathbb{Z}^{d_i}$ vers un ensemble de valeurs. Elles représentent des flots de données ;
- $\vec{x}$ est un vecteur de $\mathbb{Z}^{d_i}$;
- les F_i sont des fonctions quelconques à n arguments ;
- les $G_{i,j}$ sont des fonctions de dépendance de $\mathbb{Z}^{d_i}$ vers $\mathbb{Z}^{d_j}$.

Notons que les expressions des $g_{i,j}$ ne font pas intervenir les variables v_i ou v_j, mais seulement les indices des domaines. Le contrôle ne peut donc pas dépendre des données. Par analogie avec les réseaux de processus, nous nous plaçons donc dans le cadre d'un contrôle déterministe et sans conflit. Les systèmes d'équations récurrentes peuvent avoir des restrictions supplémentaires. Un système dont les $G_{i,j}$ sont uniquement des translations est un SERU, si les $G_{i,j}$ sont des fonctions affines alors on parlera de SERA, *etc.*

Exemple 4.1. La fonction calculant la suite des nombres de Fibonacci est un grand classique parmi les fonctions récursives. La suite des n premiers éléments peut s'écrire sous la forme de SERU à une seule variable v, définie sur $\mathbb{N}^*$, avec $G_{1,1}(x) = x-1$ et $G_{1,2}(x) = x-2$:

$$\forall x,\ 1 \leq x \leq 2, \qquad v(x) = 1 \tag{4.2}$$

$$\forall x,\ 3 \leq x \leq n, \qquad v(x) = v(x-1) + v(x-2) \tag{4.3}$$

Un autre exemple de restriction porte sur la nature des domaines $\mathbb{Z}^{d_i}$. Chaque point du domaine doit être défini une et une seule fois, ce qui pose le problème de la *calculabilité* du programme. La calculabilité d'un SURE non-conditionnel est décidable [128]. Par contre, la calculabilité d'un SERA aux domaines non-bornés est, dans le cas général, indécidable ; un tel SERA a l'expressivité d'une machine de Turing [125]. Pour cette raison, les différents auteurs s'imposent plus ou moins de restrictions quant aux systèmes d'équations étudiés, compromis entre expressivité du modèle et calculabilité de la solution.

Langages dédiés

Le langage *Alpha* [79, 163, 233] est un exemple de langage fonctionnel permettant d'écrire des SERA. C'est un langage déclaratif, autorisant les équations conditionnelles, où les domaines de définition sont des unions de polytopes paramétrés. Mauras [163] le décrit initialement comme «*une généralisation de la notion de flux de données telle qu'elle apparaît dans Lucid [12] et Lustre [54]*». Mais à la différence de Lustre, la dimension temporelle n'est pas connue *a priori* : c'est à l'ordonnancement de déterminer les «sens de parcours» des domaines des variables.

Exemple 4.2. Le système de l'exemple 4.1 s'écrit comme suit en Alpha :

```
var
  v : { x | 1<=x<=N } of integer
let
  v = case
    { x | 1<=x<=2 } : 1.(x->)
    { x | 3<=x<=N } : v.(x->x-1) + v.(x->x-2)
  esac
tel
```

D'autres langages ont été définis dans le même esprit de description déclarative de systèmes d'équations ; citons entre autres *Sisal* [13], et plus récemment *Alphabets* et son environnement AlphaZ [165].

Array-OL [42, 78, 100] est un langage pour la description de SERU, rejoignant les principes précédemment énoncés. À la différence des langages sus-cités, il se base sur une représentation graphique ; en cela, il rejoint des modèles tels que les graphes SDF multi-dimensionnels (*cf.* sous-section 2.2.4). Les flots de données y sont représentés par des tableaux multi-dimensionnels, toriques et non-paramétrés, et éventuellement infinis sur une dimension au plus. Si le temps n'est pas spécifié *a priori*, comme en Alpha, il est généralement implicite : la dimension infinie sert souvent à représenter, dans la pratique, une séquence de tableaux finis sur les autres dimensions.

Résumé

Un système d'équations récurrentes est une approche déclarative pour spécifier une application flot de données. Leur forme à assignation unique rend explicites les dépendances de données. Les calculs sont *a priori* massivement parallèles, mais la notion de temps n'y est pas explicitement représentée. Ils ne peuvent donc être calculés en l'état : ils doivent d'abord être ordonnancés, pour trouver un ordre d'exécution respectant les dépendances de données (ici, des dépendances de flot). Ceci pose en fait une question fondamentale sur l'utilisation des indices des variables pour exprimer les relations entre leurs instants d'évaluation, et par conséquent leurs horloges.

4.1.2 Nids de boucles

Une seconde approche pour décrire une application flot de données consiste à l'écrire dans un langage impératif, tel que *C* ou *Fortran*. À la différence des langages dédiés de la section précédente, ce sont des langages généralistes, et n'ont pas vocation à décrire des applications flots de données. Cependant, leur popularité dans le monde industriel en font *de facto* des langages pour la modélisation de telles applications, en particulier sous la forme de *nids de boucles*.

Principes de base

Un programme impératif s'exécute selon une séquence d'*instructions* : une instruction est le plus petit élément de code exécutable. Une instruction peut être *simple* (*e.g.* évaluation, affectation, appel de fonction) ou *composée* (*e.g.* structure conditionnelle *si-alors-sinon*, boucles *pour* ou *tant que*). Une instance d'une instruction simple est une *opération*. Un nid de boucles est une imbrication finie de boucles, de la forme :

pour $x_1 \leftarrow min_1$ **à** max_1 **pas** inc_1 **faire**
  ...
  pour $x_2 \leftarrow min_2$ **à** max_2 **pas** inc_2 **faire**
    ...
    pour $x_n \leftarrow min_n$ **à** max_n **pas** inc_n **faire**
      Corps
    fin pour
    ...
  fin pour
  ...
fin pour

On appelle x_k l'*itérateur* ou l'*indice* de la $k^{\text{ème}}$ boucle. Les expressions min_k et max_k sont les *bornes* de la boucle, et le scalaire inc_k est son *pas*. n est la *profondeur* de l'instruction. Un nid de boucles est dit *parfait* si toutes les instructions simples sont dans *Corps*, donc de profondeur n. Le vecteur $(x_1, ..., x_n)$ est le *vecteur d'itération*.

Par rapport à l'expressivité propre aux langages généralistes, la description d'applications flots de données demande à faire certains aménagements syntaxiques, afin de restreintre ou spécialiser leur sémantique [223]. Ces restrictions ont pour effet de délimiter des classes de programmes que nous sommes capables de modéliser par des polyèdres ou, comme nous le verrons dans le chapitre suivant, par des réseaux de processus. Citons entre autres :

- l'ajout de directives pour enrichir le code : *e.g.* forcer ou bannir des optimisations, spécialiser des entrées/sorties ou des mémoires, affiner les opérations sur les bits ;
- l'interdiction des allocations dynamiques de mémoire ;
- l'interdiction de l'usage des pointeurs autres que ceux des tableaux, *etc.*

Partie à contrôle statique

Une partie à contrôle statique (*Static Control Part* – SCoP) est une portion de code, en particulier un nid de boucles, dont le contrôle limité permet une parallélisation efficace. Une SCoP répond aux critères suivants [18, 20] :

– Les seules structures de contrôle autorisées sont des boucles *pour* et des structures conditionnelles, dont les tests portent uniquement sur des fonctions affines des boucles englobantes et des paramètres globaux ;
– Les relations entre données sont exprimées par des *références statiques*. Les données sont traitées sous forme de tableaux ; un scalaire est considéré comme un tableau à une seule case.

L'ordre dans lequel les opérations sont exécutées au sein de la SCoP est un ordre total, imposé par deux relations d'ordre. La première relation, à supposer que toutes les boucles soient séquentielles[1] et que leurs pas soient positifs[2], est l'ordre lexicographique. Il impose un ordre entre opérations de différentes itérations.

Définition 4.3 (Ordre lexicographique)**.** *L'ordre lexicographique, noté $\prec_{lex}$, est tel que pour tous vecteurs $\vec{x}$ et $\vec{y}$ dans $\mathbb{Z}^n$:*

$$\vec{x} \prec_{lex} \vec{y} \Leftrightarrow \exists i \in [\![1, n-1]\!],\ (x_1, \ldots, x_i) = (y_1, \ldots, y_i) \wedge x_{i+1} < y_{i+1} \tag{4.4}$$

La seconde relation d'ordre est l'ordre *textuel* : si deux instructions sont écrites en séquence, alors au sein de la même itération, l'opération de la première est exécutée avant celle de la seconde.

Définition 4.4 (Référence statique)**.** *Une référence est dite statique quand elle fait référence à une cellule d'un tableau grâce à une fonction d'index affine qui ne dépend que des itérateurs des boucles englobantes et de paramètres formels.*

Définition 4.5 (Partie à contrôle statique)**.** *Un ensemble maximum d'instructions consécutives avec des domaines polyédriques convexes est appelé une partie à contrôle statique.*

Girbal *et al.* ont montré que les SCoP constituent une famille de portions de code qui représente une importante part du temps de calcul dans des applications scientifiques ou de traitement du signal [99].

Résumé

Le nid de boucles privilégie un style de spécification issu des langages de programmation généralistes. Les réécritures dans une même variable sont possibles, ce qui a pour effet de rajouter des dépendances de données (dépendances de sortie et anti-dépendances). D'autre part, la progression des indices procure une interprétation temporelle plus directe que dans les systèmes d'équations récurrentes. C'est au développeur d'imposer un ordonnancement séquentiel, et de s'assurer qu'il respecte les dépendances. Cela a donc pour effet de rendre le parallélisme implicite.

4.1.3 Liens avec le modèle polyédrique

Quelque soit l'approche utilisée pour décrire une SCoP, par un système d'équations récurrentes ou par un nid de boucles, nous arrivons à un même constat : le programme demande à être transformé et/ou ordonnancé.

D'une part, un système d'équations récurrentes ne fait pas intervenir de notion de temps, comme nous l'avons déjà mentionné. L'application est *a priori* concurrente ; cependant les relations de dépendance engendrent un ordre partiel par clôture transitive. Deux opérations non-liées par cet ordre

[1] C'est le cas de la plupart des langages généralistes ; Fortran en est une exception, depuis Fortran 95, avec la boucle parallèle FORALL.

[2] Dans le cas contraire, des transformations de boucles telles que le renversement (*loop reversal*) ou la scission (*loop splitting*) permettent de s'y ramener.

sont indépendantes, et sont potentiellement exécutables de manière concurrente. C'est donc à l'*ordonnancement* de déterminer des «sens de parcours» des domaines des variables. Plus précisément, l'ordonnancement, s'il existe[3], associe à chaque opération un instant d'exécution : seules des opérations indépendantes peuvent être ordonnancées à un même instant, afin de respecter les dépendances.

D'autre part, un nid de boucles est exécutable en l'état, on part du principe que le concepteur a imposé un ordonnancement valide. Par contre, cet ordonnancement est séquentiel et ne fait intervenir aucun parallélisme. Si la SCoP est destinée à être exécutée sur une architecture parallèle, nous souhaiterions utiliser au mieux ses ressources. Cela demande donc à extraire le parallélisme intrinsèque à la SCoP, masqué par l'ordonnancement initial. Nous devons alors *transformer* la SCoP pour la réordonnancer.

Le modèle polyédrique est particulièrement utile pour l'analyse et la transformation des SCoP, de façon à en optimiser l'exécution selon certains critères (*e.g.* réduction du temps de calcul ou des tailles des mémoires). Des travaux récents étendent même l'application du modèle polyédrique aux boucles *tant que* et aux branchements conditionnels avec prédicats quelconques [5, 22].

Avant de construire le modèle polyédrique

Si la SCoP est décrite sous forme de système d'équations récurrentes, la construction du modèle polyédrique correspondant est quasi-immédiate ; c'est la façon la plus directe et la plus naturelle de décrire un tel modèle. Les langages Alpha et Alphabets, par exemple, ont été clairement définis dans ce but.

Au contraire, une SCoP décrite sous forme de nids de boucles n'est pas adaptée pour être directement transposée dans le modèle polyédrique. Comme nous l'avons mentionné précédemment, un nid de boucles présente deux différences fondamentales par rapport à un système d'équations récurrentes : (i) plusieurs valeurs peuvent être affectées à une même variable, et (ii) les opérations sont totalement ordonnées : elles respectent l'ordre lexicographique entre deux itérations, et l'ordre textuel au sein d'une itération. La SCoP doit alors subir des traitements plus poussés, pour l'amener sous forme proche d'un système d'équations. Ces traitements sont globalement les suivants :

Décomposition analytique du nid de boucles : analyse du code en entrée, et construction d'un arbre de syntaxe abstraite.

Construction d'une forme à assignation unique : le passage en forme SSA permet d'éliminer les dépendances de sortie et les anti-dépendances, en créant de nouvelles variables.

Analyse des dépendances : les dépendances entre instructions (dépendances de flot) sont construites à partir de l'analyse des références dans l'arbre de syntaxe abstraite. On en construit un graphe de dépendance réduit.

Résolution des bornes des domaines : la construction des domaines de chaque instruction (ou variable) et de chaque dépendance demande à lister les constraintes sur les indices. Les références d'indices permettent de construire un ensemble d'équations et inéquations, dont la réduction fait appel à des outils d'algèbre linéaire [85, 197].

Différents outils implantent ces transformations de code pour la génération d'un modèle polyédrique, chacun adapté à un ensemble de langages en entrée. Citons, entre autres, *Clan* (C, C++, C#, Java) [19], *LooPo* (C, C++, Fortran) [108] ou encore *MatParser* (Matlab) [132, 133].

[3] Il se peut qu'aucun ordonnancement ne soit valide pour l'application. En particulier si le système d'équations introduit des dépendances cycliques.

Et après ?

Un grand principe de compilation impose que l'ordonnancement des opérations d'un programme soit soumis à trois types de contraintes [4] :

1. le programme, après transformation, doit effectuer les mêmes opérations que le programme original ;
2. les résultats des opérations des programmes original et transformé doivent être identiques ;
3. l'ordonnancement ne doit pas demander plus de ressources que disponibles.

Les deux premiers points, à savoir la modélisation des données et de leurs dépendances, sont abordés en section 4.2. Le troisième point, traitant de l'ordonnancement dans le modèle polyédrique, est quant à lui développé en section 4.3.

4.2 Modélisation des données et des dépendances

Dans cette section, les différents concepts sont introduits suivant une adaptation des notations utilisées par Bastoul dans sa thèse [18].

4.2.1 Domaine de définition

Comme nous l'avons vu, les algorithmes dits de traitement intensif de données portent souvent sur des instructions ou variables structurées en tableaux multidimensionnels. Intuitivement, un tableau peut être assimilé à un $\mathbb{Z}$-polytope, intersection d'un polytope et du réseau standard. Le tableau et le $\mathbb{Z}$-polytope sont de mêmes dimensions, ses faces sont définies par les bornes du tableau, et les points à coordonnées entières inclus dans le $\mathbb{Z}$-polytope correspondent aux cases du tableau. Ce $\mathbb{Z}$-polytope est appelé son *domaine de définition*. À tout point à coordonnées entières du domaine de définition correspond une opération. Il s'écrit de façon générale sous la forme :

$$\mathcal{D}_S : \left\{ \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} \in \mathbb{Z}^d \times \mathbb{Q}^n \;\middle|\; \begin{pmatrix} A_{S,X} & A_{S,P} \end{pmatrix} \cdot \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} + \vec{a}_S \geq \vec{0} \right\} \tag{4.5}$$

où $\vec{x}$ sont les coordonnées d'une opération, $\vec{p}$ est un jeu de paramètres, et $A_{S,X}$ et $A_{S,P}$ sont respectivement les matrices des cœfficients rationnels des coordonnées et des paramètres.

Exemple 4.6. Un tableau multidimensionnel est de la forme :

déclarer entier constant $p_1, p_2, \ldots, p_n$
déclarer tableau entier $S(p_1, p_2, \ldots, p_n)$

La $k^{\text{ème}}$ dimension du tableau S est bornée par 1 et p_k. Le polytope correspondant est un hyper-rectangle, ayant pour équation :

$$\mathcal{D}_S = \left\{ \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} \in \mathbb{Z}^n \times \mathbb{Q}^n \;\middle|\; \begin{pmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ -1 & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -1 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \\ \vdots \\ -1 \\ 0 \end{pmatrix} \geq \vec{0} \right\} \tag{4.6}$$

Dans le cas général, le domaine de définition ne se limite pas à un hyper-rectangle ; chacune de ses faces peut être n'importe quel hyperplan de l'espace. Il est envisageable de considérer des instructions dont le domaine de définition est un polyèdre non-borné. Par exemple, si l'instruction en question est

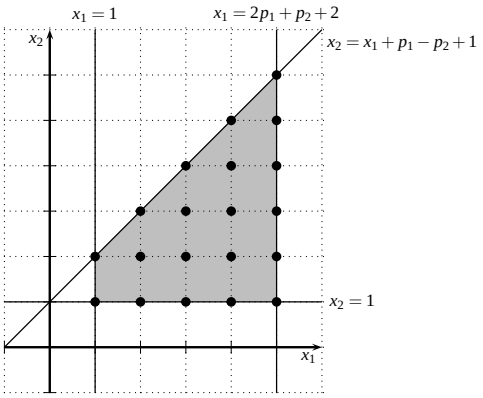


FIG. 4.1: Domaine d'itération paramétré à 2 dimensions, avec $p_1 = 1$ et $p_2 = 1$.

dans un nid de boucles dont la boucle la plus englobante est infinie. Toutefois, cela demande à ce que l'analyse de ses dépendances puisse être ramenée dans un domaine fini, et nous imposons qu'au plus une dimension soit infinie (un seul rayon dans le système générateur du polyèdre), de la même façon qu'en Array-OL.

4.2.2 Domaine d'itération

Le *domaine d'itération* d'une instruction, et en particulier d'une affectation, correspond à l'ensemble des opérations exécutées lors de l'exécution de la SCoP. Le domaine d'itération peut toujours être écrit sous la forme d'un système d'inéquations linéaires, définissant un $\mathbb{Z}$-polyèdre [138].

La profondeur de l'instruction dans le nid de boucles est égale à la dimension du $\mathbb{Z}$-polyèdre. Les bornes des boucles imposent des contraintes sur les valeurs que peuvent prendre les indices, ce qui se traduit par un $\mathbb{Z}$-polytope borné par des hyperplans ; les bornes des boucles définissent leurs équations. Les pas des boucles définissent «l'espacement» des points du $\mathbb{Z}$-polyèdre, et donc l'équation de son réseau. Si tous les pas sont unitaires, alors le $\mathbb{Z}$-polyèdre est standard.

Exemple 4.7. Considérons l'instruction S dans la SCoP suivante, avant conversion sous forme SSA :

pour $x_1 \leftarrow 1$ **à** $2p_1 + p_2 + 2$ **pas** 1 **faire**
  pour $x_2 \leftarrow 1$ **à** $x_1 + p_1 - p_2 + 1$ **pas** 1 **faire**
    $S : a(x_2) \leftarrow \ldots$
  fin pour
fin pour

Son domaine d'itération est représenté en FIG. 4.1 pour $\vec{p} = (1,1)$. Il est défini comme l'intersection d'un polytope avec le réseau standard, tel que :

$$\mathcal{D}_S = \left\{ \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} \in \mathbb{Z}^2 \times \mathbb{Q}^2 \;\middle|\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 2 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} + \begin{pmatrix} -1 \\ 2 \\ -1 \\ 1 \end{pmatrix} \geq \vec{0} \right\} \tag{4.7}$$

Remarque 4.8. Tout point appartenant à un quelconque domaine d'itération $\mathcal{D}'_S$ d'une affectation S doit également appartenir à son domaine de définition $\mathcal{D}_S$. Dans le cas contraire, nous calculons la valeur d'une variable non-définie, ou accédons à un espace mémoire non-réservé.

4.2.3 Dépendances de données

Dans le contexte d'une SCoP, la dépendance entre deux instructions se définit comme suit :

Définition 4.9 (Dépendance entre instructions)**.** *Une instruction S* dépend *d'une instruction R, et l'on note $R\delta S$, s'il existe des opérations $R(\vec{x_R})$ et $S(\vec{x_S})$, telles que :*

1. *$R(\vec{x_R})$ et $S(\vec{x_S})$ accèdent à la même adresse mémoire, et au moins l'une d'elles y écrit,*
2. *$\vec{x_R}$ et $\vec{x_S}$ appartiennent respectivement au domaine d'itération de R et S,*
3. *dans la SCoP séquentielle originale, $R(\vec{x_R})$ est exécutée avant $S(\vec{x_S})$.*

Cette définition aux trois types de dépendances possibles [72] : (i) *dépendance de flot* (R écrit, puis S lit), (ii) *anti-dépendance* (R lit, puis S écrit), (iii) *dépendance de sortie* (R et S écrivent). Les deux dernières sont éliminées par convertion de la SCoP en forme SSA. Dorénavant, toutes les dépendances étudiées sont des dépendances de flot.

Remarque 4.10. Bien que cela puisse tomber sous le sens, il est important de noter que si deux opérations ne sont pas dépendantes l'une de l'autre, alors elles sont *indépendantes*. C'est cette indépendance entre opérations qui constitue, comme nous le verrons dans la section suivante, une condition nécessaire à leur parallélisation.

Nous déduisons aussi de la définition précédente le concept de *$\mathbb{Z}$-polyèdre de dépendance* $\mathcal{D}_{S\delta R,n,z}$, capturant la relation de dépendance entre R et S : c'est un sous-ensemble du produit cartésien des espaces d'itération de R et S, où tout point à coordonnées entières représente une dépendance entre deux opérations. Dans un cas simple, non-paramétré, les bornes du $\mathbb{Z}$-polyèdre s'écrivent sous la forme :

$$\mathcal{D}_{R\delta S,n,z} = \left\{ \begin{pmatrix} \vec{x_S} \\ \vec{x_R} \end{pmatrix} \in \mathbb{Q}^{n+n'} \;\middle|\; D \begin{pmatrix} \vec{x_S} \\ \vec{x_R} \end{pmatrix} + \vec{d} = \left(\begin{array}{cc} F_S & -F_R \\ \hline A_S & 0 \\ 0 & A_R \\ Z_S & -Z_R \end{array}\right) \cdot \begin{pmatrix} \vec{x_S} \\ \vec{x_R} \end{pmatrix} + \left(\begin{array}{c} \vec{f_S} - \vec{f_R} \\ \hline \vec{a_S} \\ \vec{a_R} \\ \vec{z} \end{array}\right) \frac{=}{\geq} \vec{0} \right\} \quad (4.8)$$

telle que, pour une paire de références p :

1. Références des opérations : cette contrainte correspond à la correspondance entre les coordonnées des opérations référencées par $R(\vec{x_R})$ et $S(\vec{x_S})$, ce qui s'exprime par : $F_R\vec{x_R} + \vec{f_R} = F_S\vec{x_S} + \vec{f_S}$;
2. Domaines d'itération : $\vec{x_R}$ et $\vec{x_S}$ sont respectivement dans les domaines d'itération de R et S, d'où $A_R\vec{x_R} + \vec{a_R} \geq \vec{0}$ et $A_S\vec{x_S} + \vec{a_S} \geq \vec{0}$;
3. Précédence : n est un niveau de dépendance, correspondant à une profondeur d'imbrication dans les boucles ; $x_{R,i} = x_{S,i}$ si $i < n$, et $x_{R,n} < x_{S,n}$ si n est inférieur à la profondeur commune des instructions. Sinon, il n'y a pas de dépendance supplémentaire. On en construit un ensemble d'inéquations linéaires : $Z_S\vec{x_S} - Z_R\vec{x_R} + \vec{z} \geq \vec{0}$.

Par ailleurs, son réseau est l'intersection des réseaux des domaines d'itération de R et S :

$$\Lambda_{R\delta S} = \Lambda_R \cap \Lambda_S \quad (4.9)$$

Une dépendance $R\delta S$ existe si et seulement si son $\mathbb{Z}$-polyèdre $\mathcal{D}_{R\delta S,n,p}$ contient au moins un point entier ; ceci peut être vérifié par programmation linéaire en nombre entier.

Exemple 4.11. Reprenons l'exemple 4.7, que nous enrichissons de la façon suivante :

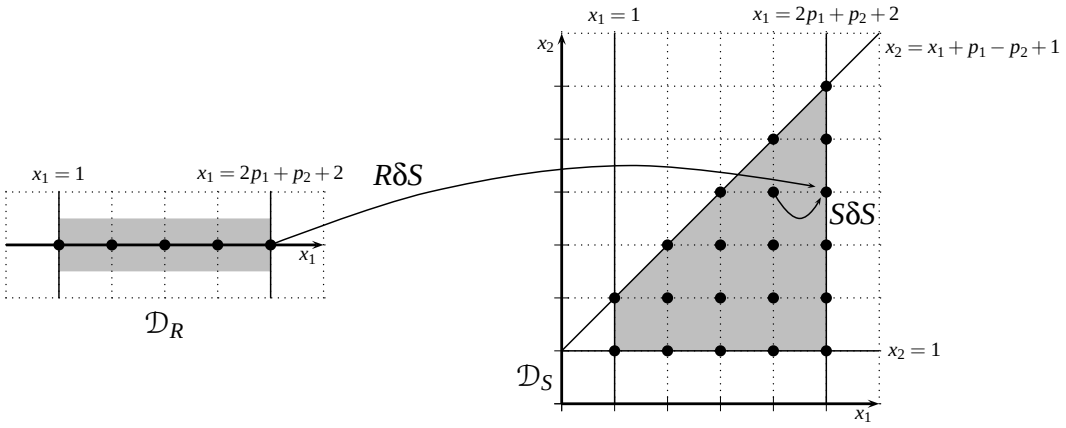


FIG. 4.2: Dépendances de l'opération $a(4,4)$, avec $p_1 = 1$ et $p_2 = 1$.

pour $x_1 \leftarrow 1$ **à** $2p_1 + p_2 + 2$ **pas** 1 **faire**
  $R : b(x_1) \leftarrow \ldots$
  pour $x_2 \leftarrow 1$ **à** $x_1 + p_1 - p_2 + 1$ **pas** 1 **faire**
    $S : a(x_2) \leftarrow f(a(x_2), b(x_1))$
  fin pour
fin pour

Cette SCoP contient trois dépendances : $\mathcal{D}_{R\delta S,1,\langle b(x_1),b(x_1')\rangle}$, $\mathcal{D}_{S\delta S,1,\langle a(x_2),a(x_2')\rangle}$ et $\mathcal{D}_{S\delta S,2,\langle a(x_2),a(x_2')\rangle}$. La FIG. 4.2 représente ces dépendances pour le calcul de l'opération $a(4,4)$. La troisième dépendance est vide, nous pouvons donc la supprimer. Les deux dépendances restantes sont une dépendance de R vers S, et une autre de S vers elle-même. Elles sont définies par :

$$\mathcal{D}_{R\delta S,1,\langle b(x_1),b(x_1')\rangle} = \left\{ \begin{pmatrix} x_1 \\ x_1' \\ x_2' \\ p_1 \\ p_2 \end{pmatrix} \in \mathbb{Z}^3 \times \mathbb{Q}^2 \;\middle|\; \left(\begin{array}{ccccc} 1 & -1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 2 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & 1 & -1 \end{array}\right) \cdot \begin{pmatrix} x_1 \\ x_1' \\ x_2' \\ p_1 \\ p_2 \end{pmatrix} + \left(\begin{array}{c} 0 \\ \hline -1 \\ 2 \\ -1 \\ 2 \\ -1 \\ 1 \end{array}\right) \underset{\geq}{=} \vec{0} \right\} \tag{4.10}$$

$$\mathcal{D}_{S\delta S,1,\langle a(x_2),a(x_2')\rangle} = \left\{ \begin{pmatrix} x_1 \\ x_2 \\ x_1' \\ x_2' \\ p_1 \\ p_2 \end{pmatrix} \in \mathbb{Z}^4 \times \mathbb{Q}^2 \;\middle|\; \left(\begin{array}{cccccc} 1 & 0 & -1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 2 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 1 & 0 & -1 & 0 & 0 \end{array}\right) \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_1' \\ x_2' \\ p_1 \\ p_2 \end{pmatrix} + \left(\begin{array}{c} 0 \\ \hline -1 \\ 2 \\ -1 \\ 1 \\ -1 \\ 2 \\ -1 \\ 1 \\ -1 \end{array}\right) \underset{\geq}{=} \vec{0} \right\} \tag{4.11}$$

Pour simplifier les notations, nous noterons par la suite $\mathcal{D}_{R\delta S}$ comme étant l'unique polyèdre de dépendance de $R\delta S$.

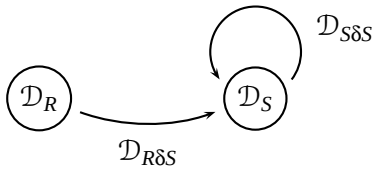


FIG. 4.3: Graphe de dépendance réduit polyédrique des instructions R et S.

4.2.4 Graphe de dépendance réduit polyédrique

Les dépendances de données sont communément représentées sous la forme d'un graphe de dépendance [4]. En particulier, dans notre cas, les dépendances de données au sein d'une SCoP sont représentées par un *graphe de dépendance réduit polyédrique* (GDRP) [72].

Définition 4.12 (Graphe de dépendance réduit polyédrique)**.** *Un graphe de dépendance réduit polyédrique est un graphe orienté. Une instruction est représentée par un unique sommet, étiqueté par son domaine d'itération. Les dépendances entre instructions sont représentées par des arcs, étiquetés par les polyèdres de dépendance.*

Exemple 4.13. Reprenons l'exemple 4.11. Le graphe de dépendance réduit polyédrique des instructions R et S est représenté en FIG. 4.3.

4.3 Ordonnancement

Maintenant que nous avons spécifié des contraintes sur les domaines des instructions et leurs dépendances, nous cherchons à les ordonnancer. Ici, l'*ordonnancement* s'entend au sens défini précédemment : il associe à chaque opération l'instant où elle est exécutée. La différence majeure vient du fait que le temps est ici un concept multidimensionnel.

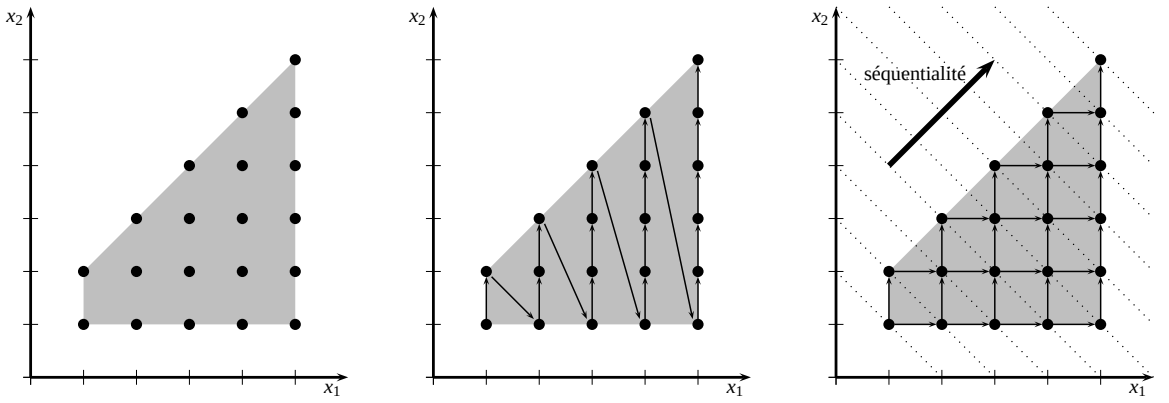
Tout d'abord, nous rappelons de façon intuitive ce que représentent les concepts de *temps* et d'*espace* dans le modèle polyédrique. Pour cela, nous procédons par analogie avec les nids de boucles. Ensuite, nous présentons formellement la méthode d'ordonnancement utilisée, et les transformations dans le modèle polyédrique. Les résultats sont principalement issus des travaux de Feautrier [86, 87], Bastoul [18] et Pouchet *et al.* [194, 195, 196].

4.3.1 Notions de temps et d'espace

En l'absence de dépendance

L'exécution d'un programme se décompose en une séquence d'instants, au même titre que l'exécution des modèles des chapitres précédents. Comme nous l'avons vu, un nid de boucles impose un ordre total sur ses opérations, selon les ordres lexicographique et textuel. Son exécution est séquentielle : selon cet ordre total, une seule opération est allouée par instant. Considèrons une instruction S imbriquée dans n boucles, sans dépendances :

pour $x_1 \leftarrow min_1$ **à** max_1 **pas** inc_1 **faire**
  ...
  pour $x_n \leftarrow min_n$ **à** max_n **pas** inc_n **faire**



(a) Les opérations sont toutes indépendantes. Elles peuvent être évaluées simultanément.

(b) Les dépendances imposent un ordre total sur les opérations, elles sont calculées en séquence.

(c) Les dépendances n'imposent qu'un ordre partiel sur les opérations ; plusieurs d'entre elles peuvent être évaluées à chaque instant.

FIG. 4.4: Différents exemples de l'incidence des dépendances de données sur l'ordonnancement.

$S(x_1,\ldots,x_n)$
fin pour
...
fin pour

Son domaine d'itération $\mathcal{D}_S$ est modélisé par un $\mathbb{Z}$-polyèdre de dimension $\dim \mathcal{D}_S = n$ dans l'espace vectoriel $\mathbb{Z}^n$. Puisque ses n dimensions sont parcourues en séquence, une seule ressource (*e.g.* un processeur) suffit au calcul de ses opérations : la ressource calcule une opération par instant.

Le fait d'introduire du parallélisme dans le calcul de S revient à introduire une nouvelle forme de boucle, **pour tout**, permettant d'exécuter en parallèle certaines de ses opérations. Ici, par *parallélisme* nous entendons *simultanéité* et non *concurrence* : la boucle **pour tout** effectue toutes ses itérations au même instant. Par exemple, nous pouvons réécrire le précédent nid de boucle sous la forme :

pour $x_1 \leftarrow min_1$ **à** max_1 **pas** inc_1 **faire**
...
pour tout $x_k \leftarrow min_k$ **à** max_k **pas** inc_k **faire**
...
$S(x_1,\ldots,x_n)$
...
fin pour
...
fin pour

Ainsi, le domaine $\mathcal{D}_S$ est parcouru en séquence sur ses dimensions 1 à $k-1$, et en parallèle sur ses dimensions k à n. Pour un sous-vecteur $(x_1,\ldots,x_{k-1})$ donné, toutes les opérations de S sont exécutées simultamément sur un ensemble de ressources parallèles.

Exemple 4.14. Considérons le nid de boucle suivant, illustré en FIG. 4.4(a) :

pour $x_1 \leftarrow 1$ **à** 5 **pas** 1 **faire**

pour $x_2 \leftarrow 1$ **à** $x_1 + 1$ **pas** 1 **faire**
$S(x_1, x_2) \leftarrow f(x_1, x_2)$
fin pour
fin pour

Puisque S ne dépend d'aucune autre instruction, on peut envisager d'effectuer toutes ses opérations simultanément :

pour tout $x_1 \leftarrow 1$ **à** 5 **pas** 1 **faire**
pour tout $x_2 \leftarrow 1$ **à** $x_1 + 1$ **pas** 1 **faire**
$S(x_1, x_2) \leftarrow f(x_1, x_2)$
fin pour
fin pour

Les deux dimensions de $\mathcal{D}_S$ sont alors parcourues dans l'*espace* (en parallèle). Alors que le premier nid de boucle demande vingt instants pour s'exécuter sur une ressource, le second ne demande qu'un seul instant à vingt ressources en parallèle.

Avec dépendances

Dans le cas général, les instructions peuvent être interdépendantes ; il peut aussi exister des dépendances entre les opérations d'une même instruction. Les ordonnancements des instructions d'une SCoP donnée ne sont *valides* que si et seulement si toute opération est exécutée *au moins* un instant après les opérations dont elle dépend. Cela pour deux raisons :

1. Le respect de la causalité implique que l'on ne peut dépendre du futur ;
2. On interdit que deux opérations interdépendantes s'exécutent simultanément.

Évidemment, il n'est pas toujours possible d'extraire du parallélisme d'une SCoP. Il se peut que les dépendances de données imposent un ordre total sur les exécutions des opérations.

Exemple 4.15. Considérons le nid de boucles suivant, illustré en FIG. 4.4(b) :

pour $x_1 \leftarrow 1$ **à** p **pas** 1 **faire**
pour $x_2 \leftarrow 1$ **à** $x_1 + 1$ **pas** 1 **faire**
si $x_1 = 1 \wedge x_2 = 1$ **alors**
$S(x_1, x_2) \leftarrow init$
sinon si $x_1 \geq 2 \wedge x_2 = 1$ **alors**
$S(x_1, x_2) \leftarrow f_1(S(x_1 - 1, x_1))$
sinon si $x_1 \geq 2$ **alors**
$S(x_1, x_2) \leftarrow f_2(S(x_1, x_2 - 1))$
fin si
fin pour
fin pour

La figure montre clairement que toute opération dépend, par transitivité, de l'ensemble de ses prédécesseurs. Aucun parallélisme n'est envisageable ; les deux dimensions de S doivent impérativement être parcourues en séquence. Elles sont donc *temporelles*.

Exemple 4.16. Au contraire, considérons le nid de boucles suivant, illustré en FIG. 4.4(c) :

pour $x_1 \leftarrow 1$ **à** 5 **pas** 1 **faire**
  pour $x_2 \leftarrow 1$ **à** $x_1 + 1$ **pas** 1 **faire**
    si $x_1 = 1 \wedge x_2 = 1$ **alors**
      $S(x_1, x_2) \leftarrow init$
    sinon si $x_2 = x_1 + 1$ **alors**
      $S(x_1, x_2) \leftarrow f_1(S(x_1, x_2 - 1))$
    sinon si $x_1 \geq 2 \wedge x_2 = 1$ **alors**
      $S(x_1, x_2) \leftarrow f_2(S(x_1 - 1, x_2))$
    sinon si $x_1 \geq 2 \wedge x_2 \geq 2$ **alors**
      $S(x_1, x_2) \leftarrow f_3(S(x_1 - 1, x_2), S(x_1, x_2 - 1))$
    fin si
  fin pour
fin pour

Sur la figure, il apparaît que chaque opération de S ne dépend pas de l'ensemble de ses prédécesseurs. Il existe plusieurs «sens de parcours» du domaine de S permettant d'introduire du parallélisme. Par exemple, toutes les opérations appartenant à un même segment en pointillé sont mutuellement indépendantes ; elles peuvent donc être exécutées simultanément. Cela revient à effectuer un simple changement de base sur les vecteurs d'itération pour obtenir un nouveau nid de boucles :

pour $x_1 \leftarrow 1$ **à** 10 **pas** 1 **faire**
  pour tout $x_2 \leftarrow \max(1, x_1 - 4)$ **à** $\min\left(x_1, \left\lfloor \frac{x_1}{2} \right\rfloor + 1\right)$ **pas** 1 **faire**
    si $x_1 = 1 \wedge x_2 = 1$ **alors**
      $S(x_1 - x_2 + 1, x_2) \leftarrow init$
    sinon si $x_2 = \frac{x_1}{2} + 1$ **alors**
      $S(x_1 - x_2 + 1, x_2) \leftarrow f_1(S(x_1 - x_2, x_2 - 1))$
    sinon si $x_1 \geq 2 \wedge x_2 = 1$ **alors**
      $S(x_1 - x_2 + 1, x_2) \leftarrow f_2(S(x_1 - x_2, x_2))$
    sinon si $x_1 \geq 3 \wedge x_2 \geq 2$ **alors**
      $S(x_1 - x_2 + 1, x_2) \leftarrow f_3(S(x_1 - x_2, x_2), S(x_1 - x_2, x_2 - 1))$
    fin si
  fin pour
fin pour

Par ce changement de base, nous avons associé une dimension à du *temps* (boucle séquentielle sur x_1), et l'autre à de l'*espace* (boucle parallèle sur x_2). Le temps de calcul est divisé par deux (de vingt à dix instants), tandis que le nombre de ressources nécessaires est multiplié par trois (de une à trois ressources).

Résumé

L'ordre total des opérations, imposé par un nid de boucles séquentielles, peut être plus restreint que l'ordre partiel imposé par les dépendances de données. La recherche de parallélisme au sein d'une SCoP revient à calculer un ordre d'exécution, ou *ordonnancement*, mettant en avant davantage de parallélisme. Cette recherche demande à déterminer quelles dimensions *doivent* être parcourues dans le temps, et réciproquement, lesquelles *peuvent* être parcourues dans l'espace. Intuitivement ce problème peut être vu comme un changement de base, comme illustré par l'exemple 4.16.

4.3.2 Ordonnancements valides

D'un point de vue algébrique, l'ordonnancement revient à associer de nouvelles coordonnées temporelles à une opération, pour déterminer à quel instant (itération d'une boucle **pour**), elle doit être exécutée. Pour des raisons de calculabilité, nous nous limitons à des ordonnancements affines : c'est le seul cas où nous pouvons décider de la correction des transformations, et où nous pourrons exporter la solution vers d'autres modèles, y compris générer du code.

Ordonnancement affine

Ainsi, pour toute instruction S, de domaine d'itération $\mathcal{D}_S$ et de dimension d, nous recherchons une fonction d'ordonnancement θ_S de la forme [18] :

$$\theta_S : \begin{cases} \mathbb{Z}^d \to \mathbb{Z}^n \\ \vec{x_S} \mapsto T_S \vec{x_S} + \vec{t_S} \end{cases} \qquad \text{avec } T \in \mathbb{Q}^{n \times d} \text{ et } \vec{t} \in \mathbb{Q}^n \tag{4.12}$$

θ_S associe à tout point de $\mathcal{D}_S$, et donc à toute opération de S, un vecteur représentant ses nouvelles coordonnées temporelles : un instant est donc «étiqueté» par ce vecteur. Par abus de langage, nous pouvons alors dire qu'à $\mathbb{Z}^n$ correspond un espace de temps logique. Cet espace peut être unidimensionnel $(n = 1)$ [196] ou multidimensionnel $(n > 1)$ [195]. S'il n'est pas toujours possible de trouver un ordonnancement affine à une dimension (*cf.* exemple 4.15), il en existe toujours[4] un en dimensions multiples [87].

Ensemble des ordonnancements valides

Évidemment, les dépendances entre opérations imposent des restrictions quant aux valeurs de n, T_S et $\vec{t_S}$. Pour toute dépendance $R\delta S$, θ_R et θ_S sont valides si et seulement si toute opération de R, dont dépend une opération de S, est exécutée à un instant précédant ladite opération de S : autrement dit, si $\theta_R(\vec{x_R}) \prec_{lex} \theta_S(\vec{x_S})$. Le plus petit intervalle de temps entre l'exécution des deux opérations correspond au vecteur $(0, \ldots, 0, 1)$. Cela se traduit par un ensemble d'inéquations linéaires, en éclatant la relation sur les différentes dimensions. Ces inéquations bornent l'espace des solutions possibles pour θ_R et θ_S :

$$\forall i \in [\![1, n]\!], \quad T_{S,i,*}\vec{x_S} + \vec{t}_{S,i} - T_{R,i,*}\vec{x_R} - \vec{t}_{R,i} - \delta \geq 0 \quad \text{avec } \delta = \begin{cases} 1 & \text{si } i = n \\ 0 & \text{sinon} \end{cases} \tag{4.13}$$

L'inconvénient est que le système d'inéquations ainsi généré, pour l'ensemble des dépendances du GDRP, est potentiellement (très) grand. Il peut-être réduit par application du lemme de Farkas [86].

Lemme 4.17 (Forme affine du lemme de Farkas)**.** *Soit* $\mathcal{D} = \left\{ \vec{x} \in \mathbb{Q}^d \;\middle|\; A\vec{x} + \vec{b} \geq 0 \right\}$ *un polyèdre non-vide. Une fonction affine* $f(\vec{x})$ *est non-négative partout sur* $\mathcal{D}$ *si et seulement si elle est une combinaison affine positive de la forme :*

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}\left(A\vec{x} + \vec{b}\right), \quad \textit{avec } \lambda_0 \geq 0 \textit{ et } \vec{\lambda} = \vec{0} \tag{4.14}$$

λ_0 *et* $\vec{\lambda}$ *sont appelés* multiplieurs de Farkas.

[4] Si, bien entendu, la SCoP est dépourvue de dépendances cycliques.

On obtient alors, pour toute dépendance $R\delta S$, un ensemble d'équations tel que :

$$\forall i \in [\![1,n]\!], \qquad T_{S,i,*}\vec{x_S} + \vec{t}_{S,i} - T_{R,i,*}\vec{x_R} - \vec{t}_{R,i} - \delta = \lambda_0 + \vec{\lambda}\left(D\begin{pmatrix}\vec{x_S}\\ \vec{x_R}\end{pmatrix} + \vec{d}\right) \tag{4.15}$$

avec :

$$\mathcal{D}_{R\delta S} = \left\{ \begin{pmatrix}\vec{x_S}\\ \vec{x_R}\end{pmatrix} \in \mathbb{Z}^{n+n'} \;\middle|\; D\begin{pmatrix}\vec{x_S}\\ \vec{x_R}\end{pmatrix} + \vec{d} \underset{\geq}{=} \vec{0} \right\} \tag{4.16}$$

Les multiplieurs de Farkas sont ensuite supprimés par l'élimination de Fourier-Motzkin. Ne reste qu'un ensemble d'inéquations affines, où les cœfficients et ordonnées à l'origine de la transformations dépendent uniquement de ceux des dépendances. Dans le polyèdre ainsi obtenu, tout point à coordonnées entières correspond à un ordonnancement valide. S'il n'en contient aucun, ou que la seule solution est le vecteur nul, alors la SCoP contient une erreur, comme par exemple une référence à une donnée non-calculée ou une dépendance cyclique. Ceci peut facilement être vérifié par un solveur tel que PIP [84].

4.3.3 Calcul d'un ordonnancement optimal

Il convient enfin de rechercher, parmi l'ensemble des ordonnancements valides, le «meilleur» ordonnancement *par rapport* à une métrique donnée.

Choix de la métrique

Le choix d'un «meilleur» ordonnancement est une notion arbitraire, qui dépend entièrement du critère testé, et donc de la définition de la métrique. Cette métrique est exprimée sous la forme d'une fonction affine. Puis, un solveur de programme linéaire en nombres entiers calcule un ordonnancement optimal par rapport à cette métrique, et sous l'ensemble de contraintes précédemment calculé.

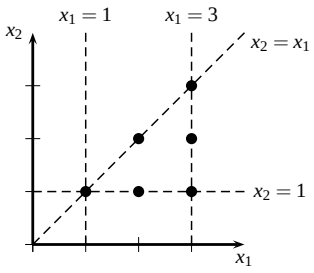
Citons, par exemple, la minimisation des latences [93], la minimisation des défauts de cache [18, 109], ou encore la minimisation des tailles de mémoire [226]. Dans ce manuscrit, nous n'imposons pas la métrique à utiliser, et laissons son choix à la discrétion de l'utilisateur. En effet, ce choix dépend de critères et contraintes qui sortent du cadre de notre étude : nature de l'application, caractéristique à optimiser et arbitrage (*e.g.* débit, énergie, nombre de ressources utilisées, tailles des mémoires), architecture cible (*e.g.* ASIC, DSP, FPGA). La littérature contient de nombreux autres algorithmes pour calculer de telles fonctions d'ordonnancement [32, 87, 105, 109, 152, 153, 182, 225].

Ajout de dimensions

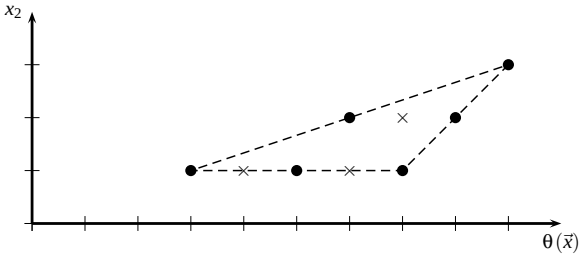
Précédemment, nous avons fait une analogie entre ordonnancement et changement de base. Le changement de base d'un domaine d'itération $\mathcal{D}$ revient à expliciter, parmi ses dimensions, lesquelles correspondent à du *temps* (parcours séquentiel), et lesquelles correspondent à de l'*espace* (parcours parallèle). Le problème que pose un changement de base et qu'il introduit, dans le cas général, des «trous» dans le polyèdre résultant : des points à coordonnées entières n'ayant pas de pré-image par la transformation. De plus, certaines transformations affines, et en particulier les projections, ne sont pas inversibles dans le cas général.

Exemple 4.18. Considérons l'exemple du polyèdre suivant [18], représenté en FIG. 4.5(a) :

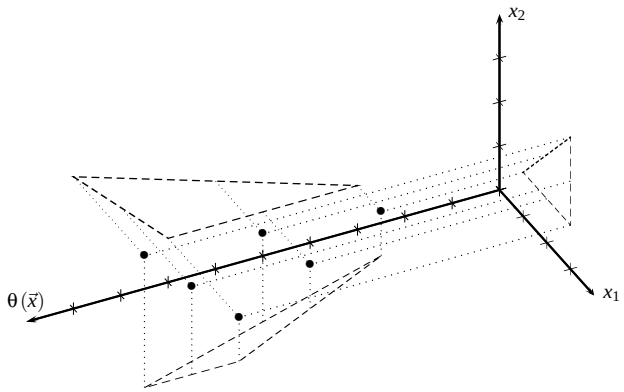
$$\mathcal{D} : \left\{ \begin{pmatrix}x_1\\ x_2\end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \begin{pmatrix}1 & 0\\ -1 & 0\\ 0 & 1\\ 1 & -1\end{pmatrix} \cdot \begin{pmatrix}x_1\\ x_2\end{pmatrix} + \begin{pmatrix}-1\\ 3\\ -1\\ 0\end{pmatrix} \geq \vec{0} \right\} \tag{4.17}$$



(a) Polyèdre d'origine $A\vec{x} + \vec{b} \geq \vec{0}$.



(b) Polyèdre après changement de base de la forme $AT^{-1}\vec{x} + \vec{b} \geq \vec{0}$.



(c) Polyèdre après ajout d'une dimension d'ordonnancement.

FIG. 4.5: Différence entre un changement de base et un ajout de dimension pour l'ordonnancement $\theta(\vec{x}) = 2x_1 + x_2$.

et l'ordonnancement $\theta(\vec{x}) = 2x_1 + x_2$. θ est une projection non-inversible, car $T = \begin{pmatrix} 2 & 1 \end{pmatrix}$. De façon à la rendre inversible, elle peut être étendue par $T = \left(\begin{smallmatrix} 2 & 1 \\ 0 & 1 \end{smallmatrix}\right)$. θ est alors un changement de base, et l'équation du polyèdre résultant est la suivante :

$$\mathcal{D}' : \left\{ \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \begin{pmatrix} 1/2 & -1/2 \\ -1/2 & 1/2 \\ 0 & 1 \\ 1/2 & -3/2 \end{pmatrix} \cdot \begin{pmatrix} \vec{x'}_1 \\ \vec{x'}_2 \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \\ -1 \\ 0 \end{pmatrix} \geq \vec{0} \right\} \tag{4.18}$$

avec $\vec{x'} = (\theta(\vec{x}), x_2)$.

Comme le montre la FIG. 4.5(b), des «trous» sont injectés dans le polyèdre en coordonnées $(4,1)$, $(6,1)$ et $(7,2)$.

La solution proposée par Bastoul [18] consiste, non pas à changer la base de l'espace, mais à ajouter aux coordonnées des opérations des dimensions supplémentaires, à savoir les coordonnées de leurs ordonnancements. Ces dimensions supplémentaires portent l'information de la transformation, sans pour autant altérer les informations initiales. Autrement dit, cette solution considère toutes les dimensions de $\mathcal{D}$ comme de l'*espace*, et y ajoute de nouvelles dimensions de *temps*. De façon générale,

nous pouvons donc écrire, pour toute instruction S :

$$\mathcal{D}'_S : \left\{ \begin{pmatrix} \vec{x'} \\ \vec{x} \\ \vec{p} \end{pmatrix} \in \mathbb{Z}^{d'+d+n} \;\middle|\; \left(\begin{array}{c|cc} I & -T_{S,X} & -T_{S,P} \\ \hline \mathbf{0} & A_{S,X} & A_{S,P} \end{array}\right) \cdot \begin{pmatrix} \vec{x'} \\ \vec{x} \\ \vec{p} \end{pmatrix} + \begin{pmatrix} \vec{t}_S \\ \vec{a}_S \end{pmatrix} \begin{matrix} = \\ \geq \end{matrix} \vec{0} \right\} \tag{4.19}$$

avec :

$$\mathcal{D}_S \quad : \quad \left\{ \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} \in \mathbb{Z}^{d+n} \;\middle|\; \begin{pmatrix} A_{S,X} & A_{S,P} \end{pmatrix} \cdot \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} + \vec{a}_S \geq \vec{0} \right\} \tag{4.20}$$

$$\vec{x'} \quad = \quad \theta_S(\vec{x}) \tag{4.21}$$

Exemple 4.19. L'autre solution, représentée en FIG. 4.5(c), ajoute une troisième dimension au polyèdre résultant, sans trou, tel que :

$$\mathcal{D}' : \left\{ \begin{pmatrix} x' \\ x_1 \\ x_2 \end{pmatrix} \in \mathbb{Z}^3 \;\middle|\; \begin{pmatrix} 1 & -2 & -1 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} x' \\ x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \\ 3 \\ -1 \\ 0 \end{pmatrix} \begin{matrix} = \\ \geq \end{matrix} \vec{0} \right\} \tag{4.22}$$

avec $x' = \theta(\vec{x})$.

4.3.4 Raffinement de l'ordonnancement

Il est parfois utile d'appliquer des transformations sur le GDRP, portant à la fois sur les domaines d'itérations et les dépendances, afin de raffiner les ordonnancements des instructions. Nous en citerons deux particulièrement adaptées :

Scission de l'ensemble d'indices (index-set splitting) : Un polyèdre peut toujours être décomposé en une union de sous-polyèdres. En particulier, le domaine d'itération d'une instruction peut-être efficacement séparé en sous-ensembles distincts de façon à faciliter ou améliorer l'ordonnancement des opérations. Griebl *et al.* ont montré que cette transformation est, dans certains cas, indispensable à l'extraction de parallélisme entre opérations [107]. Au contraire, cette méthode peut être utilisée pour réduire le nombre d'opérations d'une instruction exécutées simultanément. Prenons l'exemple trivial du système d'équations suivantes, donc les dépendances sont représentées en FIG. 4.6(a) :

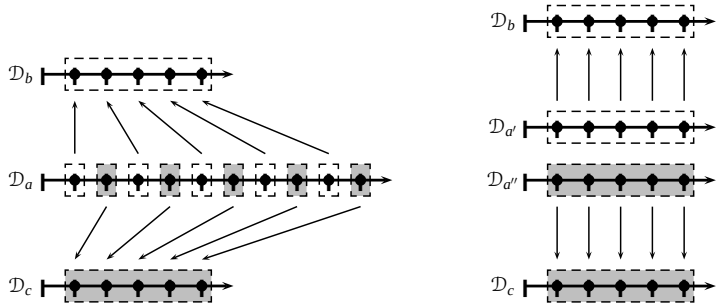
$$\begin{cases} b(x) \leftarrow f(a(2x-1), \ldots) \\ c(x) \leftarrow g(a(2x), \ldots) \end{cases} \tag{4.23}$$

Il est évident que le domaine de a peut être séparé en deux sous-polyèdres. Puis par changement d'indice (*cf.* FIG. 4.6(b)), nous obtenons :

$$\begin{cases} b(x) \leftarrow f(a'(x), \ldots) \\ c(x) \leftarrow g(a''(x), \ldots) \end{cases} \tag{4.24}$$

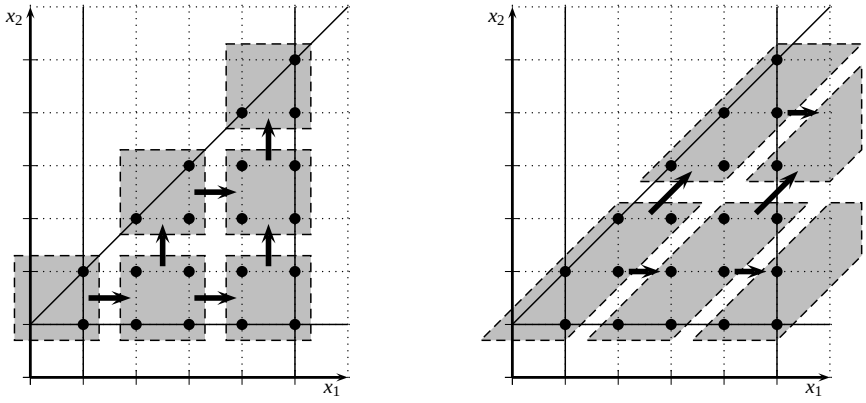
Ainsi, les calculs des points de $\mathcal{D}_{a'}$ et $\mathcal{D}_{a''}$ peuvent être ordonnancés indépendamment.

Pavage (tiling) : Bastoul [18] et Griebl *et al.* [106] ont montré l'intérêt de combiner le calcul d'ordonnancement avec du pavage. Le pavage consiste à découper le domaine d'itération en tranches plus fines, des *tuiles*, généralement pour localiser les données et améliorer la réutilisation de caches (pavage *spatial*). Le pavage peut également être *temporel* : un ensemble d'opérations, allouées à un même instant, est découpé en tuiles exécutées en séquence. Les dimensions des tuiles peuvent être paramétrées selon le nombre de ressources disponibles, voire leur position dans le domaine d'itération (*cf.* FIG. 4.7).



(a) (b)

FIG. 4.6: Exemple de scission de l'ensemble d'indices.



(a) (b)

FIG. 4.7: Exemples de pavage.

4.3.5 Discussion

Transformations et ordonnancement

La discussion sur le choix de la fonction de linéarisation à utiliser pour la construction du KRG, en sous-section 5.2.1, est un aperçu des limitations du modèle polyédrique.

Le modèle polyédrique nous permet d'effectuer des transformations complexes à un programme de traitement intensif de données. Il offre une vision ensembliste des opérations d'une instruction, associées aux points entiers inclus dans un $\mathbb{Z}$-polyèdre ; cela nous permet d'appliquer les transformations directement à un ensemble d'opérations, plutôt que de les considérer individuellement, réduisant d'autant la complexité des calculs. Il est relativement facile d'utilisation, puisqu'il s'appuie sur des outils bien connus d'algèbre linéaire et de géométrie euclidienne.

Cependant, vues sous un autre angle, ces qualités s'avèrent être des restrictions. Seules les applications affines nous permettent de rester dans le cadre du modèle polyédrique ; en particulier, le modèle polyédrique nous borne à la recherche d'ordonnancements linéaires. Pourtant, des transformations et ordonnancements optimaux dans le cadre linéaire ne le sont pas forcément dans le cadre général. Il est donc intéressant de coupler le modèle polyédrique avec un modèle à granularité plus fine, nous apportant un autre éclairage sur les résultats des transformations et ordonnancements des polyèdres.

Les KRG trouvent leur place dans ce contexte : partant des propriétés étudiées au chapitre 3, ils

nous permettent de raffiner les gestions du temps et des ressources. Les transformations locales de la topologie du graphe, aussi bien pour rerouter des jetons que pour dupliquer ou fusionner des nœuds de calcul, nous donnent la possibilité d'effectuer des optimisations locales et ponctuelles portant, par exemple, sur une unique opération, et non un sous-ensemble d'entre elles. D'un autre côté, de nombreuses optimisations s'expriment simplement sur des réseaux de processus, et sont inadaptées pour des modèles d'aussi haut niveau que le modèle polyédrique. Citons comme exemple les techniques de resynchronisation de circuits (*circuit retiming*) [147, 148], largement utilisées en synthèse logique pour optimiser le placement des registres, et qui s'appliquent naturellement aux SKRG.

Contraintes de ressources

Dans la pratique, il est inutile d'extraire plus de parallélisme de la SCoP que l'architecture, sur laquelle sera exécutée l'application finale, ne pourra en supporter. Autrement dit, le calcul de l'ordonnancement doit prendre en compte une représentation, même succinte, des contraintes induites par les ressources. Au plus simple, il s'agit d'un nombre maximal de nœuds de calcul disponibles pour exécuter un certain type d'opération (*e.g.* synthèse d'ASIC). Au plus complexe, il s'agit d'une description complète de l'architecture (*e.g.* synthèse sur FPGA, et *a fortiori* sur système-sur-puce). Dans ce manuscrit, nous nous limitons au cas simple.

Chapitre 5

Construction du graphe à routage k-périodique

> Il est plus facile de changer la spécification pour correspondre au programme que la réciproque.
>
> Alan Perlis, *Epigrams on Programming*, 1982.

Sommaire

Un flot de conception est structuré selon différents niveaux d'abstraction, et procède par raffinement, du modèle le plus global aux modèles les plus fins. Rappelons qu'un système sans conflit a un unique comportement, *modulo* l'ordre dans lequel sont exécutées les opérations (*cf.* sous-section 2.1.2). Ce comportement, asynchrone, est décrit au niveau du modèle polyédrique. Après raffinement et ordonnancement, la sémantique du modèle de plus bas niveau doit être compatible avec ce comportement.

En introduction, nous avons décrit un tel flot de conception. Partant de langages de programmation impératifs et généralistes, auxquels sont imposées certaines restrictions, le compilateur en construit

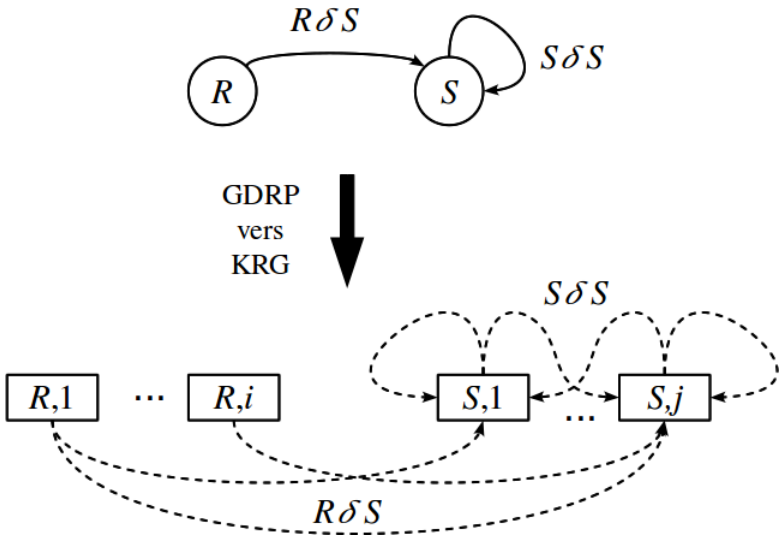


FIG. 5.1: Construction d'un KRG à partir d'un GDRP.

un modèle formel, sur lequel sont appliquées transformations et optimisations. Le principe fut l'objet de nombreux travaux, traduisants des nids de boucles en modèles polyédriques.

Nous voulons revisiter cette approche à la lumière des possibilités des KRG ; bien que les notions de temps et d'espace soient *a priori* différentes entre modèle polyédrique et KRG, nous effectuons une traduction de l'un vers l'autre et montrons les liens qui les unissent : un jeton du KRG modélise le résultat d'une opération, effectuée par une ressource, c'est-à-dire le nœud de calcul. Aussi, pour toute dépendance du le GDRP, il existe un ensemble de chemins entre couples producteurs/consommateurs. L'aspect global du KRG correspondant est représenté en FIG. 5.1.

Mais tandis que le modèle polyédrique se limite à des transformations linéaires, nous souhaitons profiter de l'expressivité des routages et ordonnancements *k*-périodiques pour élargir la classe des solutions possibles, et raffiner les performances des solutions (*e.g.* débit, dimensionnement des mémoires, *etc*). De plus, la fine granularité d'un KRG nous permet d'exploiter à la fois parallélisme de données (*data parallelism*) et parallélisme d'instruction (*instruction-level parallelism*).

Par souci de simplicité, les transformations de ce chapitre sont exprimées pour des domaines d'itération et des dépendances modélisées par des polytopes ($\mathbb{Z}$-polytopes sont le réseau est $\mathbb{Z}^n$). La formulation dans le cas de $\mathbb{Z}$-polytopes quelconques repose sur les mêmes principes, et ne demande qu'à prendre en compte les expressions de leurs réseaux. Notons cependant que nous ne perdons pas en généralité, puisque pour tout $\mathbb{Z}$-polytope, il est possible de le compresser en un polytope (*cf.* annexe A.2).

5.1 Construction des nœuds de calcul

Dans le chapitre 4, nous avons vu comment construire un GDRP et ordonnancer ses opérations. Au niveau du GDRP, nous ne disposons que de propriétés sémantiques et fonctionnelles, issues de l'*application*, pour effectuer les transformations. Maintenant, nous voulons allouer ses opérations aux ressources qui vont les exécuter, à savoir les nœuds de calcul du KRG. Le KRG est vu ici comme un modèle de plus bas niveau, grâce auquel nous allons modéliser un circuit électronique correspondant à la SCoP.

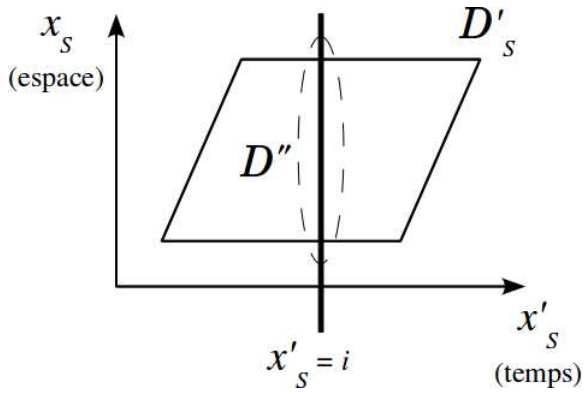


FIG. 5.2: Intersection entre le domaine d'itération $\mathcal{D}'_S$ et l'hyperplan $\vec{x'}_S = \vec{i}$.

La construction du KRG nous demande donc d'intégrer des contraintes supplémentaires, issues de l'*architecture*. En particulier, le nombre maximum de nœuds de calcul à disposition pour exécuter chaque instruction. Nous nous plaçons au niveau le plus fin, pour exploiter le parallélisme d'instruction extrait du GDRP. Deux opérations peuvent être allouées au même nœud de calcul si et seulement si elles sont allouées à des instants différents. Réciproquement, toutes les opérations devant être exécutées à un instant donné doivent être allouées à autant de ressources distinctes.

Dans une première sous-section, nous abordons le problème du dénombrement des opérations à exécuter à chaque instant, et en cherchons un majorant. Dans une seconde sous-section, nous définissons deux méthodes pour allouer les opérations aux nœuds de calcul.

5.1.1 Dénombrement des opérations par instant

Dans un GDRP quelconque, considérons une instruction S d'ordonnancement θ_S, et un vecteur de paramètres $\vec{p}$. Comme nous l'avons rappelé en sous-section 4.3.3, l'ordonnancement d'une instruction ajoute des dimensions à son domaine de définition. Si les dimensions du domaine de définition $\mathcal{D}_S$ correspondent aux dimensions *spatiales* des opérations de S, alors nous pouvons l'inclure dans un polyèdre $\mathcal{D}'_S$: les dimensions supplémentaires sont celles de l'ordonnancement. Nous avons alors :

$$\theta_S(\vec{x}_S) = T_{S,X} \cdot \vec{x}_S + T_{S,P} \cdot \vec{p} + \vec{t}_S \tag{5.1}$$

$$\mathcal{D}_S \stackrel{\text{def}}{=} \left\{ \begin{pmatrix} \vec{x}_S \\ \vec{p} \end{pmatrix} \in \mathbb{Z}^d \times \mathbb{Q}^n \;\middle|\; \begin{pmatrix} A_{S,X} & A_{S,P} \end{pmatrix} \cdot \begin{pmatrix} \vec{x}_S \\ \vec{p} \end{pmatrix} + \vec{a}_S \geq \vec{0} \right\} \tag{5.2}$$

$$\mathcal{D}'_S \stackrel{\text{def}}{=} \left\{ \begin{pmatrix} \vec{x'}_S \\ \vec{x}_S \\ \vec{p} \end{pmatrix} \in \mathbb{Z}^{d+d'} \times \mathbb{Q}^n \;\middle|\; \begin{pmatrix} I & -T_{S,X} & -T_{S,P} \\ \mathbf{0} & A_{S,X} & A_{S,P} \end{pmatrix} \cdot \begin{pmatrix} \vec{x'}_S \\ \vec{x}_S \\ \vec{p} \end{pmatrix} + \begin{pmatrix} -\vec{t}_S \\ \vec{a}_S \end{pmatrix} \begin{matrix} = \\ \geq \end{matrix} \vec{0} \right\} \tag{5.3}$$

Nombre d'opérations par instant

L'ensemble des opérations calculées à un instant étiqueté $\vec{i}$, pour un jeu de paramètres $\vec{p}$, correspond donc à l'ensemble des points $\left(\vec{x'}_S, \vec{x}_S, \vec{p}\right)$ de $\mathcal{D}'_S$ vérifiant $\vec{x'}_S = \vec{i}$. Cet ensemble est donc égal à l'ensemble des points entiers contenu dans sous-polytope $\mathcal{D}''$ de $\mathcal{D}'_S$, borné par le nouvel hyperplan $\vec{x'}_S = \vec{i}$. L'idée est illustrée par la FIG. 5.2.

Le nombre de points entiers de $\mathcal{D}''$, c'est-à-dire le nombre de points exécutés à un quelconque instant $\vec{x'}_S$, peut-être calculé grâce à un quasi-polynôme d'Ehrhart P (*cf.* annexe A.2.2). Ce nombre

de points entiers est égal à $P\left(\vec{x'}_S, \vec{p}\right)$, dont les variables sont les coordonnées de l'instant $\vec{x'}_S$ et les paramètres $\vec{p}$.

Nombre maximum d'opérations par instant

Une fois le quasi-polynôme P connu, nous pouvons rechercher le nombre maximum d'opérations effectuées simultanément, pour tout instant.

Pour cela, il faut aussi connaître l'ensemble des valeurs possibles pour $\vec{p}$. Il semble raisonnable de penser que cet ensemble de valeurs est fini : dans une application réelle, les paramètres ne sont pas choisis n'importe comment, et respectent eux-mêmes certaines contraintes. Supposons donc que l'ensemble des valeurs de $\vec{p}$ est borné, et qu'il peut être représenté par un polytope, tel que :

$$B \cdot \vec{p} + \vec{b} \geq \vec{0} \tag{5.4}$$

L'inconvénient du quasi-polynôme P est que son degré est potentiellement grand. Un calcul en nombres entiers d'une borne maximale de $P\left(\vec{x'}_S, \vec{p}\right)$ peut être très coûteux. En revanche, nous pouvons approximer la borne supérieure en passant dans le domaine réel ; nous recherchons alors un *majorant* du nombre maximum d'opérations par instant. Il est soumis à un ensemble de contraintes, issu des systèmes (5.3) et (5.4) :

$$\left(\begin{array}{ccc} I & -T_{S,X} & -T_{S,P} \\ \hline \mathbf{0} & A_{S,X} & A_{S,P} \\ \mathbf{0} & \mathbf{0} & B \end{array}\right) \cdot \begin{pmatrix} \vec{x'}_S \\ \vec{x}_S \\ \vec{p} \end{pmatrix} + \left(\begin{array}{c} -\vec{t}_S \\ \hline \vec{a}_S \\ \vec{b} \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0} \tag{5.5}$$

Les dimensions de $\vec{x}$ peuvent être éliminées de (5.5) par la projection de Fourier-Motzkin, de façon à ne conserver que des contraintes sur $\vec{x'}_S$ et $\vec{p}$, de la forme :

$$C \cdot \begin{pmatrix} \vec{x'}_S \\ \vec{p} \end{pmatrix} + \vec{c} \geq \vec{0} \tag{5.6}$$

Enfin, ce majorant est calculé par un solveur de programmes non-linéaires, en utilisant la méthode des multiplieurs de Lagrange et les conditions de Karush-Kuhn-Tucker [139, 205]. Sa formulation est la suivante :

$$\text{maximiser } P\left(\vec{x'}_S, \vec{p}\right) \tag{5.7}$$

sous l'ensemble de contraintes (5.6).

Discussion de la valeur du majorant

Soit n le majorant du nombre maximum d'opérations simultanées de S, donné par le programme non-linéaire (5.7). Notre modèle ne comprend pas de description précise de l'architecture cible. Une extension possible serait de tenir compte de contraintes sur les ressources : *e.g.* un maximum n_{max} du nombre de nœuds de calculs, dans le KRG cible, pouvant effectuer des opérations de S.

Autrement dit, nous cherchons à effectuer une transformation semblable à une projection, à chaque instant, d'au plus n opérations sur n_{max} nœuds de calcul. Se pose alors le problème de la réponse à apporter si $n > n_{max}$; puisque nous avons supposé qu'un nœud peut effectuer *au plus* une opération par instant, il est évident qu'aucune allocation n'est satisfaisante dans ce cas. Plusieurs pistes sont envisageables :

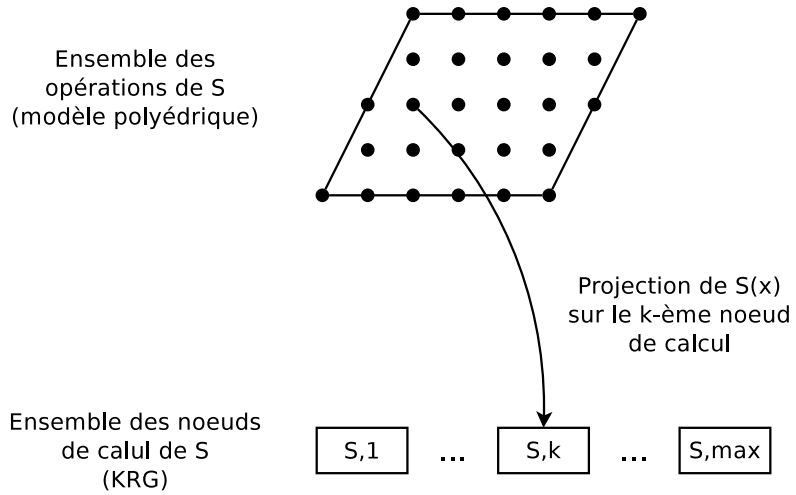


FIG. 5.3: Linéarisation des opérations : allocation aux nœuds de calcul.

– Surcontraindre l'ensemble des ordonnancements possibles (*cf.* sous-section 4.3.2) pour réduire le parallélisme.
– Adapter ou changer la métrique de l'ordonnancement (*cf.* sous-section 4.3.3) ;
– Scinder les ensembles d'opérations à exécuter par instant par pavage (*cf.* sous-section 4.3.4).

5.1.2 Linéarisation des opérations

Nous cherchons maintenant à attribuer, à chaque opération de S, le nœud de calcul chargé de la calculer. D'un point de vue algébrique, les opérations à exécuter à l'instant étiqueté $\vec{i}$ correspondent aux points entiers d'un sous-polytope de $\mathcal{D}'_S$ (système (5.3)) avec $\vec{x'}_S = \vec{i}$. Les nœuds de calcul, quant à eux, sont identifiés par des nombres entiers. Ainsi, l'allocation des opérations aux nœuds de calcul revient à effectuer une projection de $\mathbb{Z}^d$ vers $\mathbb{N}^*$ (*cf.* FIG. 5.3).

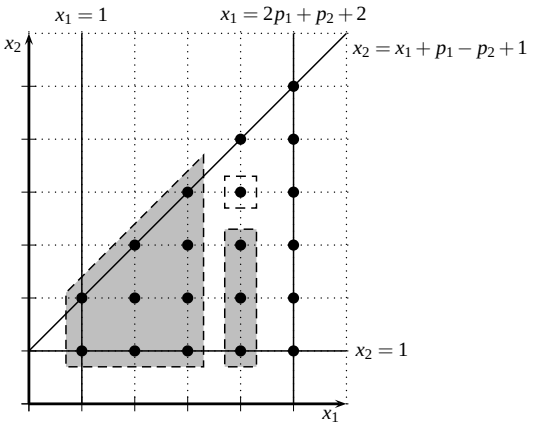
Ce problème, connu sous le nom de *linéarisation*, fut abordé par Turjan *et al.* [227] dans le contexte d'allocation de mémoire ; ils proposent différentes projections pour associer une opération à une case mémoire. Deux d'entre elles peuvent être adaptées à notre problème d'allocation de nœuds de calcul, à savoir, la *projection par le rang* et la *projection linéaire*.

Projection par le rang

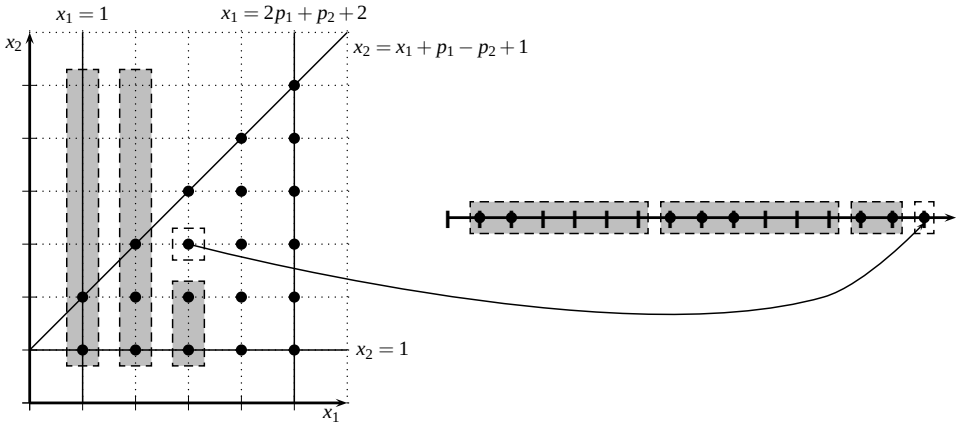
Une fonction polynôme r, dite de *rang*, associe à chaque opération de coordonnées $\vec{x}_S$ l'indice du nœud cible, numéroté entre 1 et $P\left(\vec{x'}_S, \vec{p}\right)$. La fonction de rang est de la forme :

$$r\left(\vec{x'}_S, \vec{x}_S, \vec{p}\right) = \sum_{i=1}^{d} P'_i\left(\vec{x'}_S, \vec{x}_S, \vec{p}\right) + 1 \tag{5.8}$$

Les P'_i sont des quasi-polynômes : P'_i dénombre les points entiers de $\mathcal{D}'_S$ qui, pour un $\vec{x'}_S$ et un $\vec{p}$ donnés, ont les mêmes coordonnées que $\vec{x}_S$ sur leurs $i-1$ premières dimensions, et dont la $i^{\text{ème}}$ composante est strictement inférieure à x_i (*cf.* FIG. 5.4(a)). Le rang d'une opération est alors déterminé par le nombre de points du polytope qui la précèdent selon l'ordre lexicographique. Cette projection permet de faire une projection «précise» de $P\left(\vec{x'}_S, \vec{p}\right)$ opérations sur autant de nœuds de calcul, sans



(a) Projection par le rang : le rang du point $(4,4)$ dans le polyèdre est déterminé par nombre de points d'abscisse inférieure, plus le nombre de point d'abscisse égale et d'ordonnée inférieure. Il est ici égal à 13.



(b) Projection linéaire : le nombre de points entiers lexicographiquement inférieurs à $(3,3)$ dans le polyèdre est majoré.

FIG. 5.4: Exemples de linéarisation.

insérer de trous dans l'image par r. L'inconvénient majeur de cette fonction de rang est évidemment d'être polynomiale : elle est non-inversible[1] et ne peut être exprimée dans le modèle polyédrique, qui ne permet de modéliser que des formes affines.

Projection linéaire

La projection linéaire correspond à la transformation usuelle de linéarisation de tableau multidimensionnel en tableau unidimensionnel [4, 173] : par exemple, une référence de la forme $a(x_1, x_2)$ est remplacée par $a(x_1 m + x_2)$. Étendue en dimensions quelconques, cette projection d'une opération $\vec{x}$

[1] Plus précisément, dans le cas général, r n'admet pas d'application réciproque linéaire. Cependant, c'est une bijection, elle admet donc une application réciproque r^{-1}. Celle-ci peut-être calculée en résolvant un ensemble d'équations diophantiennes de la forme $\sum_i P'_i\left(\vec{x'}_S, \vec{x}_S, \vec{p}\right) - \rho = 0$, où ρ est le rang de $\vec{x}_S$. Il est alors nécessaire d'expliciter $\vec{p}$ afin de pouvoir calculer $\vec{x}_S$.

sur un nœud de calcul d'indice i peut donc s'écrire sous la forme :

$$\vec{m}_S^T \cdot \vec{x} + m_{S,0} = i \tag{5.9}$$

L'inconvénient de cette solution est de projeter les opérations de façon discontinue sur les ressources si le polyèdre n'est pas un hyper-rectangle. C'est une application injective, comme illustrée en FIG. 5.4(b).

5.2 Construction de l'interconnexion

Cette section aborde le problème de l'interconnexion des nœuds de calcul. Connaissant la répartition des opérations sur les nœuds de calcul, nous construisons les liaisons entre producteurs et consommateurs, en veillant à respecter les dépendances.

5.2.1 Interconnexion des nœuds de calcul

Une opération, dans le modèle polyédrique, correspond à la production d'un jeton par un nœud de calcul, dans le KRG. Nous avons cité précédemment différentes méthodes pour associer une opération à un nœud de calcul ; autrement dit, nous savons par quel nœud de calcul est produit un jeton donné.

Exposé du problème

Considérons une dépendance non-vide $R\delta S$ entre deux instructions R et S du GDRP. Il existe donc un ensemble d'opérations S, de coordonnées $\vec{x}_S$, qui dépendent chacune d'une opération de R de coordonnées $\vec{x}_R$. Notons n_R et n_S les nœuds de calcul du KRG associés, respectivement, à $\vec{x}_R$ et $\vec{x}_S$ par linéarisation. Cette dépendance signifie, dans le KRG, que le jeton produit par n_R lors de l'évaluation de l'opération $\vec{x}_R$ sera routé vers n_S, pour que celui-ci puisse effectuer l'opération $\vec{x}_S$. Entre n_R et n_S, le jeton correspondant à l'opération de $\vec{x}_R$ ne peut franchir que des nœuds de sélection, de fusion, ou de copie. Autrement dit, une dépendance entre opérations du GDRP se traduit, dans le KRG, par des chemins entre leurs producteurs et leurs consommateurs. La construction de ses chemins soulève trois questions :

Transport : comment construire l'interconnexion du KRG (arcs, places, routage) de façon à acheminer tout jeton de son producteur à son consommateur ?

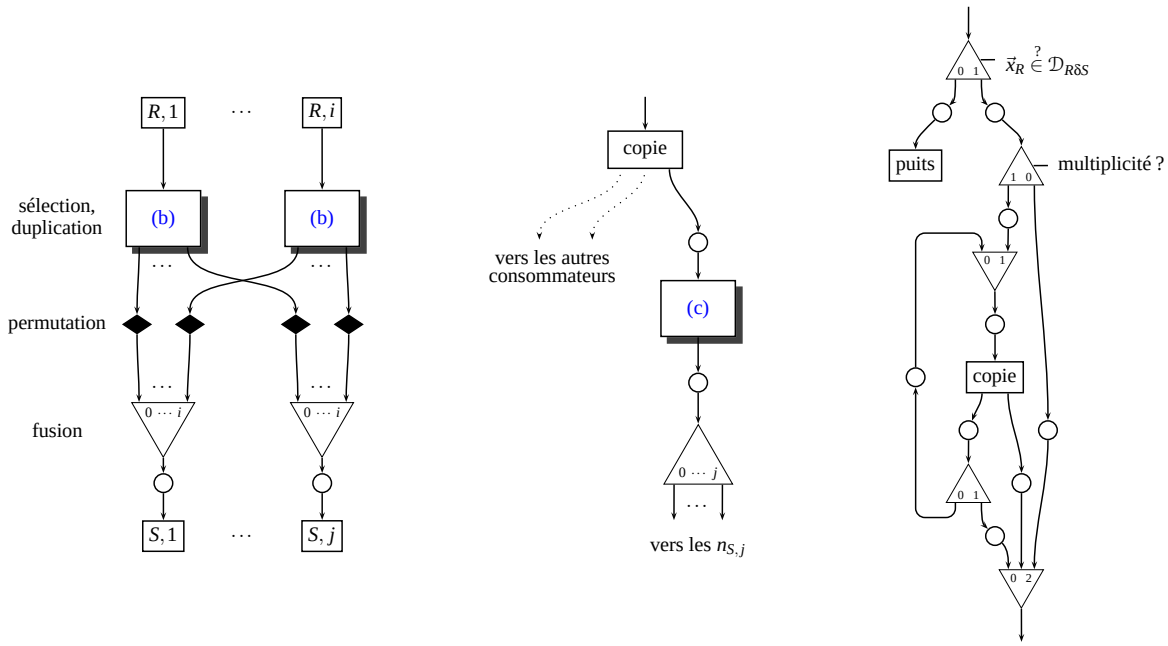
Multiplicité : si une opération est référencée plusieurs fois, comment dupliquer son jeton autant que nécessaire ?

Ordre : si, pour un couple producteur/consommateur donné, les jetons ne sont pas consommés selon leur ordre de production, comment les permuter ?

Impact de la linéarisation

Le choix de la fonction de linéarisation des opérations a un impact sur la construction de l'interconnexion. Nous devons faire un choix entre la *précision* de la projection par le rang, et la *souplesse* de la projection linéaire.

Si nous optons pour une projection par le rang, nous optimisons l'allocation des opérations aux ressources, dans le sens où nous projetons efficacement n opérations sur n ressources indicées sur $[\![1,n]\!]$. En revanche, la construction du réseau d'interconnexion et des permutations devient beaucoup plus complexe : nous sortons du domaine de compétence des outils d'algèbre linéaire, et la plupart



(a) Vue globale du routage entre producteurs $n_{R,i}$ et consommateurs $n_{S,j}$.

(b) Détail de la duplication et de la sélection de jetons vers l'ensemble des consommateurs.

(c) Détail de l'élimination ou de la duplication de jetons vers les seuls consommateurs $n_{S,j}$.

FIG. 5.5: Routage des jetons entre l'ensemble des producteurs de l'instruction *R*, et ses consommateurs pour l'instruction *S*.

des calculs ne peuvent être réalisés que par la force brute (*e.g.* expliciter les paramètres, énumérer les opérations et les traiter individuellement).

Au contraire, si nous choisissons la projection linéaire, la projection sur les ressources est plus grossière, mais la caractérisation du routage et des permutations reste paramétrique. Dans le cas particulier où les polyèdres sont des hyper-rectangles, les deux projections sont équivalentes.

Pour ces raisons, nous ne détaillons dans la suite de cette section que le cas de la projection linéaire. Le fait que les calculs restent paramétrés demande à raisonner sur la façon de construire l'interconnexion selon les paramètres. La construction de l'interconnexion pour une projection par le rang est couteuse en temps de calcul (*i.e.* considérer individuellement chaque opération et construire son routage), mais ne présente pas de difficulté particulière.

Forme générale de l'interconnexion

Nous considérons toujours une dépendance non-vide $R\delta S$ du GDRP. L'interconnexion entre les nœuds calculant les opérations de *R*, et les nœuds calculant les opérations de *S*, est illustrée en FIG. 5.5. Cette interconnexion a la forme, *a priori*, d'un «*crossbar*» entre les producteurs de *R* et les consommateurs de *S* : il existe potentiellement un chemin allant de tout nœud $n_{R,i}$ (calcul de *R* et indice *i*) vers tout nœud $n_{S,j}$ (calcul de *S* et indice *j*). Nous raffinerons par la suite cette hypothèse en supprimant les chemins inutiles, grâce aux propriétés sur les séquences de routage et les opérateurs binaires.

Dans l'idée, la séquence de jeton issue d'un producteur $n_{R,i}$ est éclatée et, connaissant l'allocation des nœuds de calcul aux opérations de *S*, chaque jeton est routé vers le consommateur approprié. Rappelons que l'interconnexion doit satisfaire aux problèmes de transport, de multiplicité et d'ordre,

introduits précédemment. Plus en détails, la structure de l'interconnexion est donc la suivante :

1. Sélections et fusions des jetons entre producteurs et consommateurs sont réalisées par des nœuds de n-fusion et de n-sélection. Le nombre d'entrées de la fusion est égal au nombre de nœuds de calcul de R, tandis que le nombre de sorties de la sélection est égal au nombre de nœuds de calcul de S. Entre les deux, des trieurs réordonnent les jetons (*cf.* FIG. 5.5(a)) ;
2. Avant la sélection, un étage de copie duplique les jetons vers chaque consommateur potentiel, donc pour chaque dépendance issue de R. Par exemple, s'il existe une autre dépendance $R\delta T$, alors les jetons correspondant aux opérations de R sont dupliqués et routés à la fois vers les nœuds de S et de T (*cf.* FIG. 5.5(b)) ;
3. Toutes les opérations de R ne sont pas forcément nécessaires aux opérations de S. Puisque les copies inutiles ne peuvent pas être directement supprimées, elles sont routées vers des puits (*cf.* FIG. 5.5(c)). Au contraire, si plusieurs opérations de S dépendent d'une même opération de R, alors le jeton correspondant est dupliqué par une boucle de copie.

Remarquons que cette interconnexion à une forme expansée : en sous-section 3.3.4, nous avons défini une normalisation de l'interconnexion sans duplication de jeton, et nous avons montré que toute interconnexion sans copie pouvait s'y ramener (*cf.* théorème 3.54). Ici, nous nous plaçons dans le cas le plus général, avec copies (*cf.* conjecture 3.55).

Les sections suivantes détaillent la construction des routages des différents points.

5.2.2 Validité et copie

Dans cette sous-section, nous déterminons de façon paramétrée le routage du troisième point ci-dessus (*cf.* FIG. 5.5(c)). Tout jeton correspondant à une opération $\vec{x}_R$, et produit par un nœud $n_{R,i}$, est systématiquement routé vers les nœuds de S. Il s'agit donc de supprimer ces jetons lorsqu'ils ne sont pas nécessaires, ou les dupliquer s'ils ont plusieurs consommateurs.

Suppressions et copies

Par l'équation (5.9), ces jetons vérifient :

$$\vec{m}_R^T \cdot \vec{x}_R + m_{R,0} - i = 0 \tag{5.10}$$

Puisque l'ensemble des valeurs possibles de $\vec{x}_R$ est aussi borné par $\mathcal{D}_R$, nous obtenons le système suivant :

$$\left(\begin{array}{ccc} \vec{m}_R^T & \vec{0}^T & -1 \\ \hline A_{R,X} & A_{R,P} & 0 \end{array}\right) \cdot \begin{pmatrix} \vec{x}_R \\ \vec{p} \\ i \end{pmatrix} + \begin{pmatrix} m_{R,0} \\ \hline \vec{a}_R \end{pmatrix} \underset{\geq}{=} \vec{0} \tag{5.11}$$

Les points $\vec{x}_R$ qui vérifient ce système sont donc les coordonnées des opérations dans la limite du domaine d'itération, et allouées au nœud $n_{R,i}$. Ainsi, il suffit de vérifier, pour tout $\vec{x}_R$ vérifiant (5.11), combien il existe d'opérations de S de coordonnées $\vec{x}_S$ dépendant de $\vec{x}_R$ dans $\mathcal{D}_{R\delta S}$. Rappelons que dans le cas paramétré (raffinement de (4.8)), la dépendance $R\delta S$ impose les contraintes suivantes :

$$\left(\begin{array}{cccc} F_{S,X} & F_{S,P} & -F_{R,X} & -F_{R,P} \\ \hline A_{S,X} & A_{S,P} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & A_{R,X} & A_{R,P} \\ Z_{S,X} & Z_{S,P} & -Z_{R,X} & -Z_{R,P} \end{array}\right) \cdot \begin{pmatrix} \vec{x}_S \\ \vec{q} \\ \vec{x}_R \\ \vec{p} \end{pmatrix} + \begin{pmatrix} \vec{f}_S - \vec{f}_R \\ \hline \vec{a}_S \\ \vec{a}_R \\ \vec{z} \end{pmatrix} \underset{\geq}{=} \vec{0} \tag{5.12}$$

L'ensemble des points $\vec{x}_S$ que nous recherchons est alors obtenu en combinant les ensembles de contraintes (5.11) et (5.12) :

$$\left(\begin{array}{ccccc} \vec{0}^T & \vec{0}^T & \vec{m}_R^T & \vec{0}^T & -1 \\ F_{S,X} & F_{S,P} & -F_{R,X} & -F_{R,P} & 0 \\ \hline A_{S,X} & A_{S,P} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0} & \mathbf{0} & A_{R,X} & A_{R,P} & 0 \\ Z_{S,X} & Z_{S,P} & -Z_{R,X} & -Z_{R,P} & 0 \end{array}\right) \cdot \begin{pmatrix} \vec{x}_S \\ \vec{q} \\ \vec{x}_R \\ \vec{p} \\ i \end{pmatrix} + \left(\begin{array}{c} m_{R,0} \\ \vec{f}_S - \vec{f}_R \\ \hline \vec{a}_S \\ \vec{a}_R \\ \vec{z} \end{array}\right) \underset{\geq}{=} \vec{0} \tag{5.13}$$

Les points $\vec{x}_S$ qui respectent (5.13), pour un $\vec{x}_R$ donné, sont ensuite dénombrés en fonction de $\vec{p}$, $\vec{q}$ et i. Ce dénombrement est obtenu par le calcul du quasi-polynôme correspondant, que nous notons P. Pour tout $\vec{x}_R$ associé à la ressource i, la valeur de $P(\vec{x}_R, \vec{p}, \vec{q}, i)$ nous indique comment router le jeton :

- Si $P(\vec{x}_R, \vec{p}, \vec{q}, i) = 0$, aucune opération de S ne dépend du jeton de $\vec{x}_R$, il est alors supprimé (routé vers le puits) ;
- Si $P(\vec{x}_R, \vec{p}, \vec{q}, i) = 1$, alors le jeton de $\vec{x}_R$ est routé vers un nœud $n_{S,j}$ sans être copié. Il existe un unique $\vec{x}_S$ vérifiant cet ensemble de contraintes, alloué à la ressource d'indice j tel que $j = \vec{m}_S^T \cdot \vec{x}_S + m_{0,S}$;
- Si $P(\vec{x}_R, \vec{p}, \vec{q}, i) > 1$, alors il existe plusieurs consommateurs de l'opération $\vec{x}_R$. Son jeton correspondant est dupliqué en $P(\vec{x}_R, \vec{p}, \vec{q}, i)$ instances, chacune étant routée vers un nœud $n_{S,j}$, et dont les indices sont déterminés comme au point précédent, selon les $\vec{x}_S$ vérifiant les contraintes.

Ordre de production des jetons

Finalement, connaissant l'ordonnancement de chaque opération de coordonnées $\vec{x}_R$ (équation (5.1)), il suffit de ranger les instants selon l'ordre lexicographique pour construire les séquences de routage appropriées des sélections et fusions. Pour cela, nous raffinons les contraintes du système (5.13) pour associer à chaque $\vec{x}_R$ son instant de production $\vec{x'}_R = \theta_R(\vec{x}_R)$:

$$\left(\begin{array}{cccccc} I & -T_{R,X} & -T_{R,P} & \mathbf{0} & \mathbf{0} & 0 \\ \vec{0}^T & \vec{m}_R^T & \vec{0}^T & \vec{0}^T & \vec{0}^T & -1 \\ \mathbf{0} & -F_{R,X} & -F_{R,P} & F_{S,X} & F_{S,P} & 0 \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & A_{S,X} & A_{S,P} & 0 \\ \mathbf{0} & A_{R,X} & A_{R,P} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0} & -Z_{R,X} & -Z_{R,P} & Z_{S,X} & Z_{S,P} & 0 \end{array}\right) \cdot \begin{pmatrix} \vec{x'}_R \\ \vec{x}_R \\ \vec{p} \\ \vec{x}_S \\ \vec{q} \\ i \end{pmatrix} + \left(\begin{array}{c} -\vec{t}_R \\ m_{R,0} \\ \vec{f}_S - \vec{f}_R \\ \hline \vec{a}_S \\ \vec{a}_R \\ \vec{z} \end{array}\right) \underset{\geq}{=} \vec{0} \tag{5.14}$$

L'énumération des points entiers de (5.14) selon l'ordre lexicographique (*i.e.* selon $\vec{x'}_R$) nous donne les productions des opérations de coordonnées $\vec{x}_R$ au cours du temps.

Bilan

En résumé, nous connaissons ou avons calculé :

- Quels sont les jetons, correspondant aux opérations de R, produits par le nœud de calcul $n_{R,i}$;
- L'ordre dans lequel ces jetons sont produits au cours du temps ;
- Si le jeton d'une opération de coordonnées $\vec{x}_R$ doit être supprimé ou non ;
- Si le jeton d'une opération coordonnées $\vec{x}_R$ doit être copié ou non, et si oui, combien de copies.

Nous avons alors tous les éléments pour construire le routage de la FIG. 5.5(c).

5.2.3 Routage de la *n*-sélection

La construction du routage du nœud de *n*-sélection de la FIG. 5.5(b) découle directement de l'étape précédente. Pour tout jeton produit par le nœud de calcul $n_{R,i}$, nous pouvons calculer l'ensemble des opérations de S qui en dépendent. D'un point de vue algébrique, pour tout $\vec{x}_R$ vérifiant (5.14), nous en déduisons le sous-polytope contenant l'ensemble des $\vec{x}_S$ correspondant. À toute opération de coordonnées $\vec{x}_S$ est associé un nœud de calcul $n_{S,j}$, tel que, par l'équation (5.9) :

$$\vec{m}_S^T \cdot \vec{x}_S + m_{S,0} = j \tag{5.15}$$

Si $\vec{x}_{S,1}, \ldots, \vec{x}_{S,k}$ est la suite des coordonnées des opérations de S obtenue par énumération, alors le routage de la *n*-sélection est de la forme :

$$\left[\left(\vec{m}_S^T \cdot \vec{x}_{S,1} + m_{S,0}\right), \ldots, \left(\vec{m}_S^T \cdot \vec{x}_{S,k} + m_{S,0}\right)\right] \tag{5.16}$$

5.2.4 Routage de la *n*-fusion

La construction du routage du nœud de *n*-fusion de la FIG. 5.5(a) est très similaire aux calculs des précédentes sous-sections.

Soit j l'indice d'un quelconque nœud de calcul $n_{S,j}$ de S. Les opérations $\vec{x}_S$ allouées à $n_{S,j}$ vérifient (5.15). L'ensemble des jetons des opérations $\vec{x}_R$, dont dépendent les opérations $\vec{x}_S$ allouées à $n_{S,j}$, et leurs instants de consommation $\vec{x'}_S$ (égaux aux instants de production des $\vec{x}_S$), sont donc solutions du système suivant :

$$\left(\begin{array}{cccccc} I & -T_{S,X} & -T_{S,P} & \mathbf{0} & \mathbf{0} & 0 \\ \vec{0}^T & \vec{m}_S^T & \vec{0}^T & \vec{0}^T & \vec{0}^T & -1 \\ \mathbf{0} & F_{S,X} & F_{S,P} & -F_{R,X} & -F_{R,P} & 0 \\ \hline \mathbf{0} & A_{S,X} & A_{S,P} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & A_{R,X} & A_{R,P} & 0 \\ \mathbf{0} & Z_{S,X} & Z_{S,P} & -Z_{R,X} & -Z_{R,P} & 0 \end{array}\right) \cdot \left(\begin{array}{c} \vec{x'}_S \\ \vec{x}_S \\ \vec{q} \\ \vec{x}_R \\ \vec{p} \\ j \end{array}\right) + \left(\begin{array}{c} -\vec{t}_S \\ m_{S,0} \\ \vec{f}_S - \vec{f}_R \\ \hline \vec{a}_S \\ \vec{a}_R \\ \vec{z} \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0} \tag{5.17}$$

De la même façon qu'à l'étape précédente, les points $\vec{x}_R$ qui vérifient (5.17) correspondent aux jetons de R consommés par le nœud de calcul $n_{S,j}$ au cours du temps, pour effectuer les opérations de coordonnées $\vec{x}_S$.

La séquence de routage de la *n*-fusion est alors obtenue par énumération, selon l'ordre lexicographique (sur $\vec{x'}_S$, donc au cours du temps), des points vérifiant (5.17). $\vec{x}_{R,1}, \ldots, \vec{x}_{R,k}$ les composantes relatives à $\vec{x}_R$, dans (5.17), alors la séquence de routage est de la forme :

$$\left[\left(\vec{m}_R^T \cdot \vec{x}_{R,1} + m_{R,0}\right), \ldots, \left(\vec{m}_R^T \cdot \vec{x}_{R,k} + m_{R,0}\right)\right] \tag{5.18}$$

5.2.5 Permutations

Connaissant comment les jetons sont routés *vers* et *depuis* un nœud de calcul, il ne nous reste plus qu'un problème à résoudre : si ces jetons sont produits puis consommés dans des ordres différents, nous devons les permuter. Dans cette sous-section, nous allons caractériser le trieur chargé de réordonner les jetons entre les nœuds de calcul $n_{R,i}$ et $n_{S,j}$ (*cf.* sous-section 3.3.3).

Recherche des jetons permutés

Considérons deux opérations d'une instruction R, de coordonnées respectives $\vec{x}_R$ et $\vec{y}_R$, dont dépendent respectivement deux opérations d'une instruction S, de coordonnées $\vec{x}_S$ et $\vec{y}_S$. Rappelons les cinq conditions nécessaires à la permutation des jetons correspondant à $\vec{x}_R$ et $\vec{y}_R$, et leurs traductions algébriques :

1. Le jeton de $\vec{x}_R$ est produit avant celui de $\vec{y}_R$:
$$\vec{x'}_R \prec_{lex} \vec{y'}_R \tag{5.19}$$
avec $\vec{x'}_R = \theta_R(\vec{x}_R)$ et $\vec{y'}_R = \theta_R(\vec{y}_R)$;
2. Le jeton de $\vec{y}_R$ est consommé avant celui de $\vec{x}_R$:
$$\vec{y'}_S \prec_{lex} \vec{x'}_S \tag{5.20}$$
avec $\vec{x'}_S = \theta_S(\vec{x}_S)$ et $\vec{y'}_S = \theta_S(\vec{y}_S)$;
3. Les opérations $\vec{x}_R$ et $\vec{y}_R$ sont effectuées par un même nœud de calcul $n_{R,i}$:
$$\vec{m}_R^T \cdot \vec{x}_R + m_{0,R} = \vec{m}_R^T \cdot \vec{y}_R + m_{0,R} = i \tag{5.21}$$
4. Les opérations $\vec{x}_S$ et $\vec{y}_S$ sont effectuées par un même nœud de calcul $n_{S,j}$:
$$\vec{m}_S^T \cdot \vec{x}_S + m_{0,S} = \vec{m}_S^T \cdot \vec{y}_S + m_{0,S} = j \tag{5.22}$$
5. $\vec{x}_R$ et $\vec{y}_R$ suivent la même dépendance vers $\vec{x}_S$ et $\vec{y}_S$, respectivement :
$$(\vec{x}_R, \vec{x}_S), (\vec{y}_R, \vec{y}_S) \in \mathcal{D}_{R\delta S} \tag{5.23}$$

Les contraintes (5.19) et (5.20) signifient que les jetons sont produits et consommés dans des ordres différents. Les contraintes (5.21), (5.22) et (5.23), quant à elles, permettent de ne considérer que les jetons susceptibles d'être permutés, c'est-à-dire suivant le même chemin dans le KRG. Ces trois dernières contraintes s'expriment sous la forme du système d'équations et d'inéquations suivant :

$$\begin{cases}
\vec{x'}_S - T_{S,X} \cdot \vec{x}_S - T_{S,P} \cdot \vec{q} - \vec{t}_S = \vec{0} \\
\vec{y'}_S - T_{S,X} \cdot \vec{y}_S - T_{S,P} \cdot \vec{q} - \vec{t}_S = \vec{0} \\
\vec{x'}_R - T_{R,X} \cdot \vec{x}_R - T_{R,P} \cdot \vec{p} - \vec{t}_R = \vec{0} \\
\vec{y'}_R - T_{R,X} \cdot \vec{y}_R - T_{R,P} \cdot \vec{p} - \vec{t}_R = \vec{0} \\
\vec{m}_S^T \cdot \vec{x}_S - \vec{m}_S^T \cdot \vec{y}_S = 0 \\
\vec{m}_R^T \cdot \vec{x}_R - \vec{m}_R^T \cdot \vec{y}_R = 0 \\
\vec{m}_S^T \cdot \vec{x}_S - j = 0 \\
\vec{m}_R^T \cdot \vec{x}_R - i = 0 \\
F_{S,X} \cdot \vec{x}_S + F_{S,P} \cdot \vec{q} - F_{R,X} \cdot \vec{x}_R - F_{R,P} \cdot \vec{p} + \vec{f}_S - \vec{f}_R = \vec{0} \\
F_{S,X} \cdot \vec{y}_S + F_{S,P} \cdot \vec{q} - F_{R,X} \cdot \vec{y}_R - F_{R,P} \cdot \vec{p} + \vec{f}_S - \vec{f}_R = \vec{0} \\
A_{S,X} \cdot \vec{x}_S + A_{S,P} \cdot \vec{q} + \vec{a}_S \geq \vec{0} \\
A_{S,X} \cdot \vec{y}_S + A_{S,P} \cdot \vec{q} + \vec{a}_S \geq \vec{0} \\
A_{R,X} \cdot \vec{x}_R + A_{R,P} \cdot \vec{p} + \vec{a}_R \geq \vec{0} \\
A_{R,X} \cdot \vec{y}_R + A_{R,P} \cdot \vec{p} + \vec{a}_R \geq \vec{0} \\
P_{S,X} \cdot \vec{x}_S + P_{S,P} \cdot \vec{q} - P_{R,X} \cdot \vec{x}_R - P_{R,P} \cdot \vec{p} + \vec{z} \geq \vec{0} \\
P_{S,X} \cdot \vec{y}_S + P_{S,P} \cdot \vec{q} - P_{R,X} \cdot \vec{y}_R - P_{R,P} \cdot \vec{p} + \vec{z} \geq \vec{0}
\end{cases} \tag{5.24}$$

Cet ensemble de contraintes définit un polyèdre, dont les solutions $(\vec{x}_R, \vec{y}_R)$ correspondent aux jetons produits par le nœud $n_{R,i}$ et consommés par le nœud $n_{S,j}$.

Nous devons maintenant contraindre le système (5.24), de façon à ne conserver que les couples $(\vec{x}_R, \vec{y}_R)$ effectivement permutés, donc vérifiant aussi les contraintes (5.19) et (5.20). Les jetons sont donc permutés si les différences $\vec{y'}_R - \vec{x'}_R$ d'une part, et $\vec{x'}_S - \vec{y'}_S$ d'autre part, sont lexicopositives. Autrement dit, la plus petite valeur pour chacune de ces différences est le vecteur $(0, \ldots, 0, 1)^T$.

Pour effectuer cette vérification, nous utilisons la même méthode que précédemment utilisée pour la recherche des ordonnancements valides en section 4.3. Il existe un ensemble d'inéquations de la forme :

$$\vec{y'}_{R,i} - \vec{x'}_{R,i} - \delta \geq 0 \qquad \text{avec } \delta = \begin{cases} 1 & \text{si } \vec{x'}_R, \vec{y'}_R \in \mathbb{Z}^i \\ 0 & \text{sinon} \end{cases} \tag{5.25}$$

$$\vec{x'}_{S,i} - \vec{y'}_{S,i} - \delta \geq 0 \qquad \text{avec } \delta = \begin{cases} 1 & \text{si } \vec{x'}_S, \vec{y'}_S \in \mathbb{Z}^i \\ 0 & \text{sinon} \end{cases} \tag{5.26}$$

Par application du lemme de Farkas aux inéquations (5.25) et (5.26) (*cf.* lemme 4.17), nous pouvons forcer à ce que ces contraintes soient lexicopositives, en recherchant des multiplieurs de Farkas $\vec{\lambda}^T$ et λ_0 dans un ensemble d'équations de la forme :

$$\vec{y'}_i - \vec{x'}_i - \delta = \lambda_0 + \vec{\lambda}^T \cdot \left(B \cdot \begin{pmatrix} \vec{y'} \\ \vec{x'} \end{pmatrix} + \vec{b} \right) \tag{5.27}$$

où $B \cdot \begin{pmatrix} \vec{y'} \\ \vec{x'} \end{pmatrix} + \vec{b}$ correspond aux contraintes, dans un cas sur $\vec{x'}_R$ et $\vec{y'}_R$, dans l'autre sur $\vec{x'}_S$ et $\vec{y'}_S$, issues de (5.24).

Dans un second temps, nous procédons à l'élimination des multiplieurs de Farkas par projection de Fourier-Motzkin. Toujours par projection de Fourier-Motzkin, nous éliminons les dimensions de $\vec{x'}_S$, $\vec{y'}_S$, $\vec{x'}_R$, $\vec{y'}_R$, $\vec{x}_S$ et $\vec{y}_S$, pour ne garder que les contraintes entre $\vec{x}_R$ et $\vec{y}_R$, en fonction des paramètres $\vec{p}$, $\vec{q}$, i et j. Nous obtenons alors un nouveau système que nous nommerons $\mathcal{D}_{perm,R\delta S}$, vérifiant les contraintes de (5.19) à (5.23). Tout point $(\vec{x}_R, \vec{y}_R)$ dans $\mathcal{D}_{perm,R\delta S}$ correspond à un couple de jetons permutés entre les nœuds $n_{R,i}$ et $n_{S,j}$; en l'occurrence, ce sont les jetons respectivement associés aux opérations de R de coordonnées $\vec{x}_R$ et $\vec{y}_R$.

Caractérisation des permutations

Cette expression des jetons permutés sous forme de polyèdre nous permet de faire le lien avec le code de Lehmer [146] : nous pouvons caractériser les permutations de jetons entre les nœuds $n_{R,i}$ et $n_{S,j}$ sous la forme d'un code de Lehmer, nous permettant ensuite de générer la permutation à proprement parler (*cf.* annexe A.3.2). Rappelons tout d'abord que le code de Lehmer d'une permutation π s'écrit sous la forme :

$$L(\pi, i) = \text{card}\left\{ j \;\middle|\; i < j,\ \pi_i > \pi_j \right\} \tag{5.28}$$

Par analogie avec notre problème de permutation de jetons, l'ensemble $\left\{ j \;\middle|\; i < j,\ \pi_i > \pi_j \right\}$ représente l'ensemble des jetons étiquetés j, produits *après* un jeton de numéro i donné $(i < j)$, et consommés *avant* $(\pi_i > \pi_j)$. C'est précisément ce que représente $\mathcal{D}_{perm,R\delta S}$: l'ensemble des jetons $\vec{y}_R$, produits après les jetons $\vec{x}_R$, et consommés avant. Le cardinal de l'ensemble, dans l'expression

(5.28), correspond dans le modèle polyédrique à un quasi-polynôme d'Ehrhart, permettant de dénombrer les points entiers d'un polytope.

Aussi, nous pouvons calculer le quasi-polynôme P, tel que $P(\vec{x}_R, \vec{p}, \vec{q}, i, j)$ soit le nombre de jetons dépassant le jeton étiqueté $\vec{x}_R$, en fonction des paramètres $\vec{p}$, $\vec{q}$, i et j.

En énumérant les jetons $\vec{x}_R$ selon leur ordre de production, nous pouvons alors reconstruire la permutation de jetons entre les nœuds $n_{R,i}$ et $n_{S,j}$. Une fois la permutation connue, il ne reste plus qu'à construire le trieur adéquat (*cf.* sous-section 3.3.3).

5.2.6 Explicitation des paramètres

Enfin, la dernière étape de la construction du KRG consiste à expliciter les paramètres. Maintenant que la projection du GDRP sur le KRG a entièrement été exprimée selon les paramètres, il ne reste qu'à énumérer les jetons et effectuer l'ensemble des calculs :

- Les arbres de *n*-sélection et *n*-fusion peuvent être simplifiés en élaguant les branches qui ne sont traversées par aucun jeton. Ces branches sont détectées par de simples calculs sur les séquences de routages des nœuds de sélection et de fusion (*cf.* sous-section 3.3.2).
- Les nœuds de calcul superflus introduits par la projection linéaire, dont nous avons supprimé les entrées, peuvent eux aussi être supprimés. Nous itérons alors sur le graphe jusqu'à ce qu'aucune simplification ne soit possible (*cf.* sous-section 3.3.1).
- Les codes de Lehmer pour chaque couple producteur/consommateur nous permettent de calculer les permutations de jetons correspondantes. Pour chaque permutation, nous construisons le trieur approprié (*cf.* sous-section 3.3.3).
- Pour chaque instruction, ses nœuds de calcul recevant le moins de jetons peuvent être fusionnés afin de réduire le nombre de ressources nécessaires (*cf.* sous-section 3.4).

5.3 Discussion

Dans cette dernière section, nous discutons de l'adéquation entre le modèle polyédrique et les KRG. Certains algorithmes ou concepts s'expriment plus ou moins facilement dans chacun de ces modèles, et nous mettons en avant l'intérêt de les utiliser de façon combinée. Les avantages et inconvénients des deux modèles laissent la porte ouverte à de nombreuses pistes de travaux futurs pour affiner le flot de conception décrit en introduction.

5.3.1 Correction de la transformation

Montrer la correction de la transformation demande à vérifier les deux propriétés suivantes : (i) un jeton est produit par un nœud de calcul si et seulement s'il existe une opération correspondante dans le modèle polyédrique ; (ii) si une opération $S(\vec{x}_S)$ dépend d'une opération $R(\vec{x}_R)$ dans le modèle polyédrique, alors le jeton correspondant à $R(\vec{x}_R)$ (ou sa copie) est consommé par le nœud de calcul produisant le jeton correspondant à $S(\vec{x}_S)$. Le premier point établit la bijection entre les opérations du GDRP et les jetons du KRG. Ensuite, il ne reste plus qu'à montrer que les dépendances entre eux sont vérifiées.

Bijection entre opérations et jetons

Notons tout d'abord que la construction du KRG ne crée aucun jeton initial. Cela veut dire que même les constantes du programme sont modélisées comme étant des données entrantes : des jetons

produits par des nœuds de calcul qui n'ont pas d'arc entrant.

Il est facile de montrer qu'à toute opération correspond une production de jeton par un nœud de calcul. C'est le résultat de la linéarisation des opérations (*cf.* sous-section 5.1.2) : à chaque instant, toutes les opérations effectuées dans cet instant sont projetées sur des nœuds de calcul. Puisque nous procédons par énumération des opérations de chaque instruction, nous sommes certains de ne pas en «oublier».

Réciproquement, cela revient à attribuer à chaque nœud de calcul la liste des opérations correspondant aux jetons qu'il produit, ordonnée selon le temps. Il s'agit donc de montrer que tout jeton produit par le nœud de calcul correspond forcément à l'une de ces opération.

Pour qu'un jeton soit produit, le nœud doit être exécuté ; pour être exécuté, le nœud doit être activé ; pour être activé, le nœud doit disposer d'au moins un jeton sur chacune de ses entrées. Aussi, deux cas peuvent se présenter :

1. Le nœud n'a pas d'entrée : tous les jetons qu'il produit correspondent à des valeurs initiales. Le nœud peut alors être tiré autant que nécessaire, mais pas plus, sinon les équations de balances ne seraient pas respectées (*cf.* sous-sections 3.1.3 et 3.1.4) ;
2. Le nœud a au moins une entrée : le nœud ne peut produire un jeton que s'il en consomme un sur chaque entrée. Autrement dit, si un nœud de calcul produit un jeton «inutile», c'est soit parce que les nœuds en amont ont eux-mêmes produit des jetons qui ne correspondent pas à des opérations, soit que les jetons sont mal routés.

Dans le premier cas, les productions de jetons dépendent de l'ordonnancement : productions et consommations doivent être balancées. Dans le second cas, les productions des jetons dépendent de l'interconnexion. Si la construction de l'interconnexion est correcte, alors aucun nœud de calcul ne peut produire un jeton qui ne correspond pas à une opération. Cela nous amène donc au second point : montrer que l'interconnexion entre les nœuds, et en particulier le routage, respecte les dépendances du GDRP.

Respect des dépendances

Pour toute dépendance $R\delta S$ du GDRP, nous construisons un chemin allant de tout nœud calculant des opérations de R à tout nœud calculant des opérations de S (*cf.* FIG. 5.5). Dans la section 5.2, nous construisons les séquences de routages en veillant à ce que tout jeton soit routé vers ses consommateurs, et seulement eux. Les dépendances sont bien respectées.

Par conséquent, seuls les nœuds de calcul devant recevoir des jetons en reçoivent. Nous en déduisons que si les productions aux entrées sont balancées seuls les jetons correspondant à des opérations du GDRP peuvent être produits. Aussi le KRG est bien construit.

5.3.2 Avantages et inconvénients

Complexité et parallélisation des calculs

Parmi les différentes étapes que nous avons proposées pour la construction des KRG, certaines d'entre elles peuvent être assez coûteuses à réaliser. Les expressions que nous en avons donné ont principalement pour but de montrer la faisabilité de la construction du KRG et d'en expliquer les principes ; toutefois, certaines de ces expressions demandent à être optimisées en vue d'une implantation efficace.

Prenons par exemple le système (5.24) : c'est la synthèse de l'ensemble des contraintes s'appliquant sur les coordonnées des opérations étudiées. Mais pour une application un minimum complexe, le solveur doit rechercher une solution à plusieurs dizaines de dimensions, soumises à quelques centaines de contraintes. L'élimination de Fourier-Motzkin, pour ne citer qu'elle, a une complexité non-polynomiale [209] ; il est donc préférable, en pratique, d'éclater ce système en de plus petits ensembles de contraintes, en éliminant progressivement les variables inutiles, plutôt que de le faire en une fois.

Néanmoins, un avantage qu'offre la construction du KRG, à mettre au crédit du modèle polyédrique, est qu'elle se base largement sur du calcul matriciel, lui-même massivement parallélisable. Par exemple, Keßler propose une implantation parallèle de l'élimination de Fourier-Motzkin, qui en réduit le temps de calcul linéairement au nombre de processeurs [130]. Aussi, comme nous l'avons vu en section 5.2, les constructions des interconnexions entre tous couples producteur/consommateur sont indépendantes : les calculs peuvent être massivement distribués.

Paramètres

Une limitation actuelle de notre approche est de ne pas pouvoir intégrer facilement des expressions paramétrées dans le KRG. Nous avons donné dans ce chapitre des expressions paramétrées pour la génération du routage et des permutations de jetons. Pourtant, la construction du KRG à proprement parler demande à expliciter les paramètres et énumérer les opérations (donc les jetons) ; les KRG n'intègrent pas le concept de routage paramétré, contrairement à d'autres modèles tels que CDDF (*cf.* section 2.5). En cela, ce problème constitue un des principaux axes de recherches futurs sur les KRG.

Exécution infinie

Un KRG permet de modéliser des comportements infinis. Comme nous l'avons mentionné au chapitre précédent, nous pouvons considérer que le GDRP représente le comportement du système au cours d'une période, répétée à l'infini : *e.g.* un nid de boucles *pour* englobées par une boucle *tant que*. Pour généraliser notre construction, il suffit simplement de remplacer toute séquence de routage calculée dans la section 5.2, ici notée u, par u^{ω}.

Ressources dédiées et description de l'architecture

Tel qu'est construit le KRG, tout nœud de calcul est dédié à une et une seule instruction. Par exemple, les opérations allouées à un nœud de calcul $n_{S,i}$ sont exclusivement des instances d'une instruction S. Même si, en soit, cette construction n'est pas vraiment une limitation à notre solution, il pourrait être intéressant d'envisager le cas où une ressource (*e.g.* un nœud de calcul faisant office d'additionneur) est partagée entre différentes instructions.

Ce point soulève toutefois la question plus générale de la prise en compte d'une description détaillée de l'architecture cible, pour le moment absente de la construction du KRG : les transformations que nous effectuons se basent uniquement sur les contraintes de l'application, et non de l'architecture. C'est, là encore, un important sujet de travaux futurs.

5.3.3 Remontée d'informations au modèle polyédrique

Latences

Les latences de calcul et, *a fortiori*, les latences de communications, ne sont pas connues au niveau du modèle polyédrique. Les latences de calcul peuvent être déduites de la technologie de l'architecture ciblée. Les latences de communication, quant à elles, dépendent du placement et du routage sur la plateforme. La mesure des latences n'intervient que bien après la construction du KRG. D'après la méthodologie de conception insensible aux latences (*cf.* sous-section 2.2.3), les latences d'interconnexion ne sont remontées au niveau du modèle qu'après un premier placement-routage.

Connaissant l'allocation des opérations aux nœuds de calcul, et les latences séparant chaque producteur de ses consommateurs, il est possible de les faire remonter au niveau du GDRP, *a posteriori*. Une solution simple consiste à jouer sur la valeur de δ, lors du calcul de l'ordonnancement, dans les inéquations (4.13) et (4.15) afin de surcontraindre l'ensemble des ordonnancements valides : connaissant la latence entre les nœuds de calcul des instructions R et de S, nous pouvons imposer une distance minimale entre leurs opérations. Cette remontée d'information peut demander à transformer le GDRP, si par exemple les latences sont non-uniformes pour toutes les opérations d'une même instruction, par scission de l'ensemble d'indices (*cf.* sous-section 4.3.4).

Ordonnancement et métrique

De façon plus générale, il serait intéressant de faire évoluer la métrique, dont dépend le choix des ordonnancements du modèle polyédrique (*cf.* sous-section 4.3.3), selon un principe de *rétroaction*. Initialement, le GDRP est ordonnancé sans connaissance des latences. Pourtant, du choix de cet ordonnancement, et donc du parallélisme qu'il introduit, dépendent directement les circuits critiques et les débits des nœuds du KRG. Remonter ces informations au niveau du modèle polyédrique rendra d'autant plus pertinents les ordonnancements calculés sur le GDRP.

Troisième partie

Du modèle au circuit

Chapitre 6

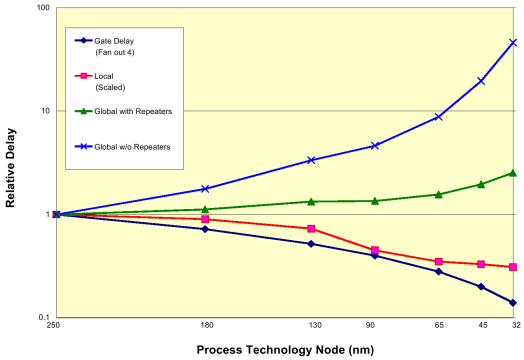
Systèmes insensibles aux latences

> En deux occasions, on m'a demandé, – De grâce, M. Babbage, si vous donnez à la machine de mauvais chiffres, les bonnes réponses en sortiront-elles ? – Dans un cas, un membre de la Chambre Haute, et dans l'autre un membre de la Basse, me posèrent cette question. Je ne suis pas capable d'appréhender, au juste, le genre de confusion d'idées qui peuvent susciter une telle question.
>
> Charles Babbage, *Passages from the Life of a Philosopher*, 1864.

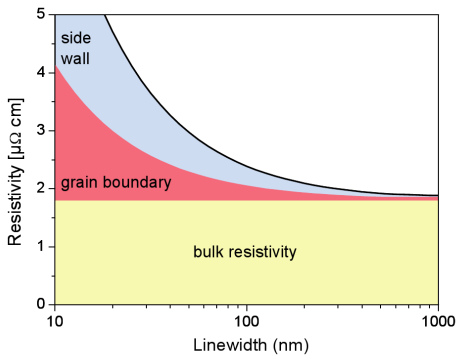
Sommaire

Dans le précédent chapitre, nous avons vu comment transformer un programme de traitement intensif de données en un modèle flot de données de type KRG. Ce chapitre détaille l'architecture du circuit intégré correspondant. Cette étape présente une difficulté : le matériel ainsi produit doit faire cohabiter contraintes physiques (en particulier les délais des signaux) et hypothèses synchrones.

En électronique numérique, les délais au sein d'un circuit intégré sont de deux natures : les délais des portes logiques, et les délais d'interconnexion. L'évolution des technologies CMOS submicroniques profondes, inférieures au quart de micromètre, impliquent une prépondérance des délais d'interconnexion sur ceux des portes. Les écarts relatifs, selon la techonologie utilisée, sont représentés en FIG. 6.1(a). En effet, les délais des portes sont réduits alors que la gravure gagne en finesse ; mais les résistivités des fils croissent, d'où une augmentation des temps de propagation des signaux [6, 16] (*cf.* FIG. 6.1(b)). Dans des circuits VLSI, plusieurs cycles d'horloge peuvent s'écouler entre l'émission et la réception d'un signal.



(a) Délais relatifs des portes *(en bleu foncé)*, des interconnexions locales *(en rouge)* et des interconnexions globales *(en vert et bleu clair)*, selon la finesse de la gravure.



(b) Résistivité des interconnexions en cuivre selon la largeur des pistes.

FIG. 6.1: Impact de la technologie de gravure sur la résistivité des fils et des délais de communication. Données ITRS [121, 122].

Ce problème doit être envisagé au plus tôt du flot de conception pour garantir la correction fonctionnelle du résultat final. Pour cela, nous appliquons la méthodologie de conception insensible aux latences, issue des travaux présentés en sous-section 2.2.3.

Dans un premier temps, nous rappelons les différentes solutions proposées dans la littérature, tout en discutant leurs implantations. Ces implantations ne sont valables que pour des flots de données purs, dépourvus de branchements conditionnels, et modélisés par des graphes marqués. Dans une deuxième partie, nous détaillons donc une implantation étendue aux systèmes de type KRG.

6.1 Architecture d'un flot de données sans routage

6.1.1 Avec protocole insensible aux latences

Dans cette sous-section, certaines notations et noms de signaux ont été modifiés et harmonisés par rapport aux articles originaux, pour conserver une cohérence entre les différentes implantations étudiées. Elles ne modifient rien fondamentalement.

Carloni *et al.*, 1999–2002

Une première implantation est décrite par Carloni *et al.* [49, 52]. La FIG. 6.2(a) donne le détail de l'implantation de la coquille[1]. La station relai et son automate sont donnés en FIG. 6.2(b) et 6.3. Les signaux de sortie, absents de l'original, ont été déduits des informations données par les auteurs.

Une donnée est un couple $(data, void)$. *data* est la donnée en elle-même, tandis que le bit *void* vaut «1» lorsque le champ *data* ne contient pas d'information (*i.e.* des bits de bourrage), et «0» si la donnée est valide. La contre-pression est assurée par le signal *stop*. La coquille filtre les données en entrée pour ne conserver que les valides. Tant que des données ne sont pas arrivées sur certaines entrées ou que *stopIn* vaut «1», l'horloge de la perle est coupée, et les données en sortie sont répétées

[1] Cette figure est une reproduction de celle donnée par les auteurs, à titre d'exemple ; la logique en est cependant incorrecte. La contre-pression (*stopOut*) est émise sur une entrée si celle-ci est vide (*voidIn*). D'autre part, le tampon doit être remplacé par un inverseur.

avec $voidOut = 1$. Lorsque toutes les données en entrée sont disponibles, la perle est exécutée. Ses résultats sont émis avec $voidOut = 0$ dès lors que la coquille ne reçoit pas de contre-pression. La contre-pression est émise sur un canal en amont si le registre en entrée contient une valeur, ou si la coquille en reçoit de l'aval.

Une station relai contient les registres pour stocker deux données. Son contrôle a deux états principaux, *Processing* et *Stalling*, alors que les états *WriteAux* et *ReadAux* sont intermédiaires et permettent de contrôler le multiplexeur et le démultiplexeur. Dans l'état initial *Processing*, une donnée reçue est écrite dans le registre principal, puis émise l'instant suivant. Si la station relai reçoit de la contre-pression, la nouvelle donnée entrante est stockée dans le registre auxiliaire. Lorsque la contre-pression cesse, le contenu du registre auxiliaire est propagé, puis on revient à l'état initial. La contre-pression reçue est toujours propagée. Ce contrôle de la station relai pose deux problèmes, qui seront discutés ultérieurement (*cf.* sous-section 6.1.3).

Casu et Macchiarulo, 2003

Casu et Macchiarulo proposent une seconde implantation du LIP [55]. Un canal est un triplet de signaux $(data, stop, val)$. La valeur d'une donnée est associée au signal *data*, tandis que sa présence correspond au signal *val* ; ce dernier est une négation des signaux *void* de Carloni *et al.*, qui eux représentent l'absence de donnée.

Bien que l'architecture de la coquille ne soit pas détaillée, on peut la considérer similaire à celle de la FIG. 6.2(a), avec de la logique ajoutée au niveau des registres. Les équations des signaux de la coquille sont les suivantes :

$$enable = \left(\bigwedge_k valIn_k\right) \wedge \left(\bigwedge_k \overline{stopIn_k \wedge valOut_k}\right) \tag{6.1}$$

$$\text{next}(valOut_i) = \left(\bigwedge_k valIn_k\right) \wedge \left(\bigwedge_{k\neq i} \overline{stopIn_k \wedge valOut_k}\right) \vee (stopIn_i \wedge valOut_i) \tag{6.2}$$

$$stopOut_i = valIn_i \wedge \left(\left(\bigvee_k stopIn_k \wedge valOut_k\right) \vee \left(\bigvee_{k\neq i} \overline{valIn_k}\right)\right) \tag{6.3}$$

La coquille peut exécuter sa perle (signal *enable*) si elle dispose de données sur chacune de ses entrées $(\forall j, valIn_j = 1)$ et si :

1. elle ne reçoit aucune contre-pression :

$$\forall k,\ stopIn_k = 0 \tag{6.4}$$

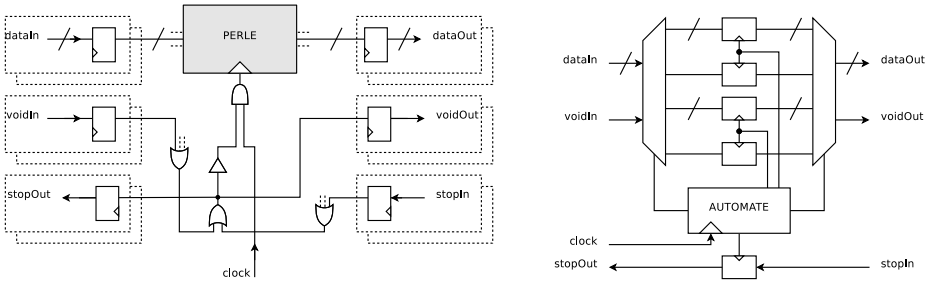
2. *ou* les sorties sur lesquelles elle en reçoit sont prêtes à stocker de nouvelles données :

$$\forall k,\ stopIn_k = 1 \ \Rightarrow\ valOut_k = 0 \tag{6.5}$$

Par conséquent, une donnée sera émise à l'instant suivant $(\text{next}(valOut_i))$ sur une sortie *i* si :

1. la perle est exécutée à l'instant présent,
2. *ou* si une donnée est déjà présente et ne peut être émise en raison de contre-pression.

On notera que l'expression est simplifiée, $\overline{stopIn_i \wedge valOut_i} \vee stopIn_i \wedge valOut_i$ étant une tautologie. Enfin, un signal *stopOut* est émis sur une entrée *i* si une donnée y est en attente, mais que la perle n'est pas exécutée.



(a) Coquille. (b) Station relai.

FIG. 6.2: Implantation de Carloni *et al.*, 1999–2002.

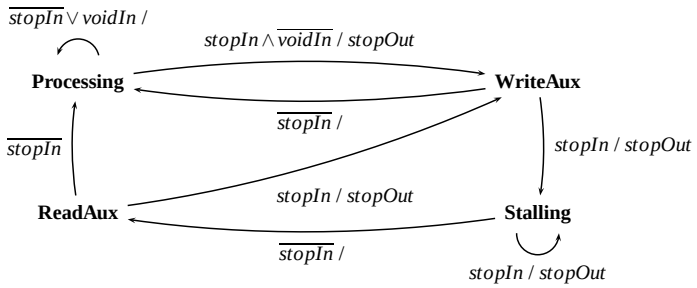


FIG. 6.3: Contrôle d'une station relai, Carloni *et al.*, 1999–2002.

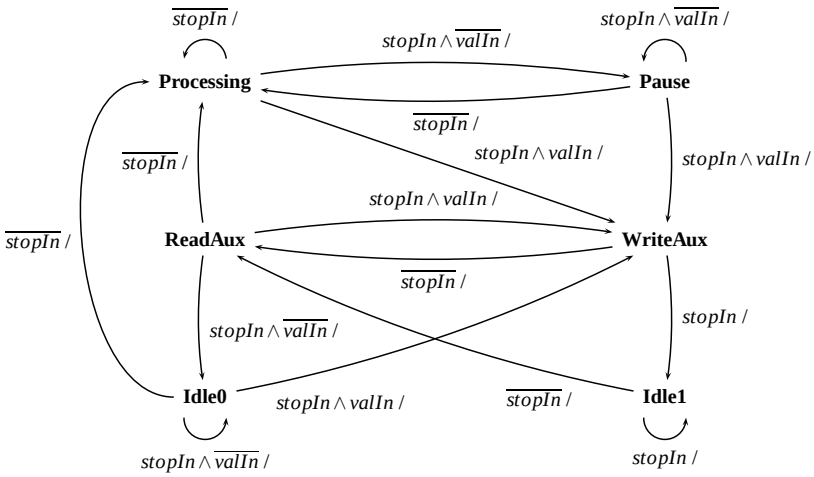


FIG. 6.4: Contrôle d'une station relai, Casu et Macchiarulo, 2003.

Une station relai est également constituée de registres en parallèles : comme précédemment, un registre auxiliaire stocke une donnée lorsque le registre principal est plein, et que la station relai reçoit de la contre-pression. Le contrôle est assuré par un automate, variante de celui de Carloni *et al.*, et reproduit en FIG. 6.4. Initialement, l'automate est en état *Processing*, et y reste tant qu'il ne reçoit pas de contre-pression. Dès lors que la station relai reçoit de la contre-pression sans recevoir de données, elle est en *Pause*. Sinon, si une donnée arrive, elle est écrite dans le registre auxiliaire dans l'état *WriteAux*. Si la contre-pression cesse, alors la donnée du registre auxiliaire est émise en *ReadAux*, sinon la station relai est stoppée en *Idle1*. Depuis *ReadAux*, si la station relai ne reçoit pas de contre pression, elle retourne dans son état initial. Sinon, selon qu'une donnée se présente ou non, elle est stockée en *WriteAux*, ou la station relai est arrêtée en *Idle0*.

Boucaron *et al.*, 2005–2007

Boucaron *et al.* proposent une troisième architecture [38, 40], plus tard étudiée par Li *et al.* [151]. Un canal est un triplet de signaux $(data, stop, val)$ comme vu précédemment. L'architecture d'une coquille est donnée en FIG. 6.5[2]. Les entrées de la coquille disposent de registres. Les auteurs font l'hypothèse qu'à la fois la coquille et sa perle sont des blocs de combinatoire, et que les temps de propagation des signaux y sont inférieurs à une période d'horloge ; raison pour laquelle les registres de données sont munis de bypass. En d'autres termes, les signaux sont supposés être capable d'aller des stations relais en amont à celles en aval en l'espace d'un cycle d'horloge. La perle s'exécute et produit des données, émettant les signaux *clock* et *valOut*, lorsque toutes les données en entrée sont présentes, et qu'aucune contre-pression n'est reçue. *stopOut* est émis lorsque la station relai en amont contient une donnée mais que *clock* vaut «0», *i.e.* la perle ne la traite pas à l'instant courant.

L'implantation d'une station relai est donnée en FIG. 6.6. Les données entrantes sont toujours stockées dans le registre principal (activation sur *valIn*). Le registre auxiliaire est utilisé pour stocker la donnée du registre principal (état $Half$) lorsque celui-ci doit accueillir une nouvelle donnée (signal *valIn*), en cas de congestion (signal *stopIn*). L'automate de contrôle est détaillé en FIG. 6.7. Son comportement est le suivant :

- La station relai est en état *Empty* lorsqu'elle est vide. La contre-pression n'est pas propagée en amont, puisqu'elle est en mesure de recevoir et stocker des données. Les horloges ou la tension peuvent être coupées jusqu'à l'arrivée d'un signal *valIn* : la donnée *dataIn* est alors écrite dans le registre principal, et l'automate passe en état *Half*.
- Dans l'état *Half*, la station relai contient une donnée dans son registre principal, et tente de la transmettre en aval. Si elle ne reçoit de signal *stopIn*, alors la donnée est considérée comme traitée par le destinataire. Selon qu'elle reçoit ou non une nouvelle donnée au cours de cet instant, elle reste en état *Half* ou retourne en état *Empty*. Si de la contre-pression est reçue mais pas de nouvelle donnée, on reste dans l'état courant. Sinon, si *stopIn* et *valIn* sont tous les deux vrais, alors la donnée du registre principal est copiée en registre auxiliaire, la donnée entrante est écrite dans le registre principal, et l'automate va en état *Full*.
- Lorsque la station relai est pleine (état *Full*), elle ne peut recevoir de nouvelle donnée, et émet *stopOut* vers le producteur en amont. L'automate reste dans cet état jusqu'à ce que *stopIn* passe à «0» ; la donnée du registre auxiliaire est alors émise, et on retourne en état *Half*.

[2] Une erreur s'était glissée dans la version originale, où le registre en entrée était activé sur $valIn \wedge clock$; ceci à cause d'un amalgamme entre l'horloge de la coquille et la condition d'activation. La figure a été corrigée de façon à ce que le *clock gating* soit dirigé par $valIn \wedge \overline{enable \oplus stored}$. On active donc l'échantillonnage sur une entrée si une donnée y est disponible, et si *enable* et *stored* ont la même valeur : il n'y a pas de donnée stockée et l'on n'exécute pas la perle, ou la donnée stockée est consommée dans l'instant et on la remplace par la donnée entrante.

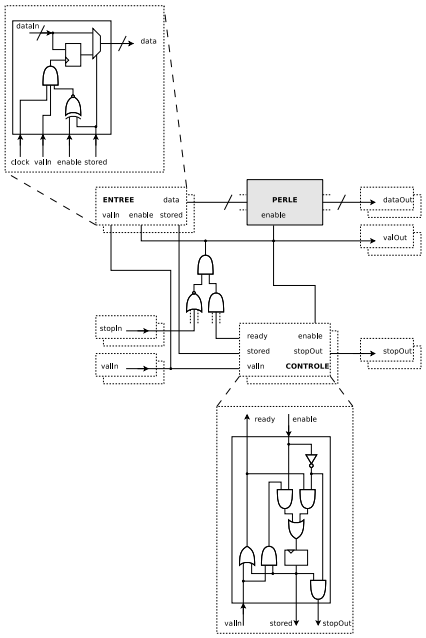
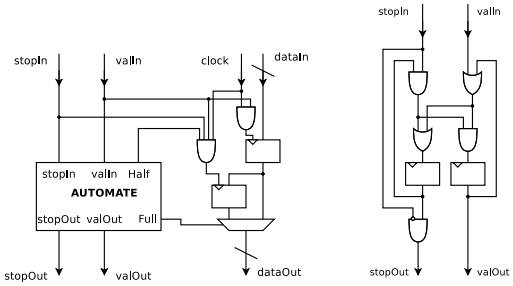


FIG. 6.5: Implantation d'une coquille, Boucaron *et al.*, 2005–2007.



(a) Vue globale.

(b) Détail de l'automate.

FIG. 6.6: Implantation d'une station relai, Boucaron *et al.*, 2005–2007.

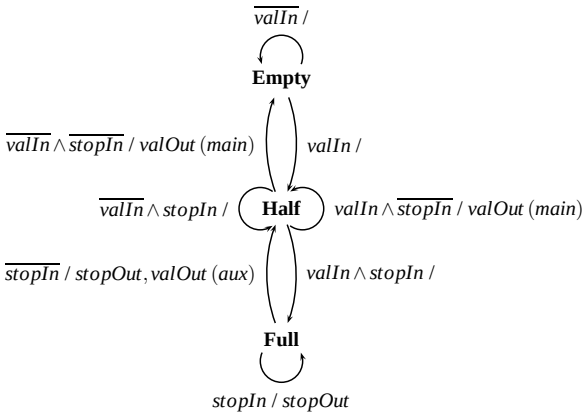
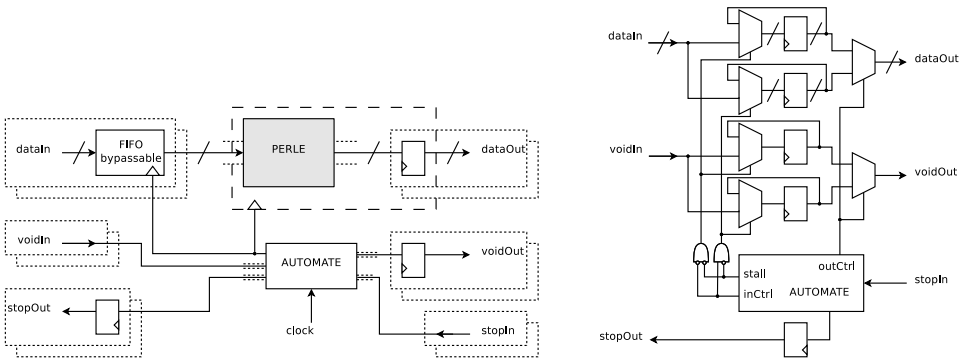


FIG. 6.7: Contrôle d'une station relai, Boucaron *et al.*, 2005–2007.



(a) Coquille. (b) Station relai.

FIG. 6.8: Implantation de Carloni, 2006.

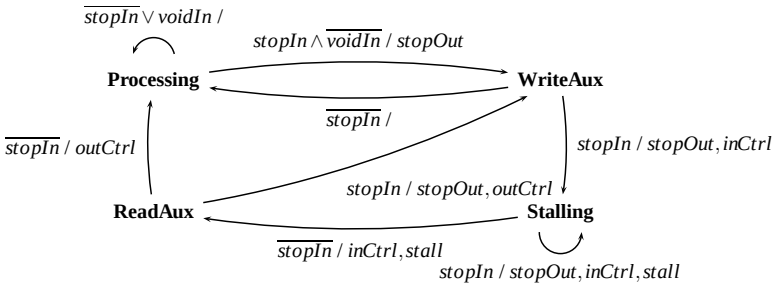


FIG. 6.9: Contrôle d'une station relai, Carloni, 2006.

Carloni, 2006

Carloni propose par la suite une solution sensiblement différente de sa première idée [48], apportant ainsi quelques corrections (*cf.* sous-section 6.1.3). L'implantation de la coquille est donnée en FIG. 6.8(a). La station relai est quant à elle détaillée en FIG. 6.8(b) et 6.9. Contrairement à la version précédente où au moins deux cycles d'horloge étaient nécessaires pour traverser la coquille, désormais celle-ci n'en impose plus qu'un au minimum, et ce afin d'éviter des franchissements combinatoires.

Trois signaux internes sont ajoutés à la description de la station relai. *inCtrl* et *outCtrl* gèrent la lecture et l'écriture dans le registre auxiliaire, tandis que *stall* coupe l'horloge des registres de façon à éviter un écrasement des données.

Cortadella *et al.*, 2006

L'implantation de Cortadella *et al.* tranche avec les précédentes par l'utilisation de bascules au lieu des flip-flops [70]. En effet, dans les solutions précédentes, un registre correspond à une flip-flop, dont nous rappelons le principe : deux bascules D en série sont nécessaires pour stocker un bit de donnée (*cf.* FIG. 6.10). La bascule maître échantillonne son entrée lorsque le signal d'horloge est bas, tandis que la bascule esclave est verrouillée. Au front d'horloge montant, la bascule maître se verrouille, tandis que la bascule esclave reçoît sa négation, donc un front d'horloge descendant ; celle-ci laisse passer le signal capturé par la bascule maître. Après une demi-période, la bascule esclave se verrouille sur ce signal pour l'émettre jusquà la fin de la période, tandis que la bascule maître laisse passer le signal en entrée, et ainsi de suite. Par exemple, l'implantation de Boucaron *et al.* (*cf.* 6.1.1) utilise $4n+4$ bascules par station relai, où n est la largeur en bits des données.

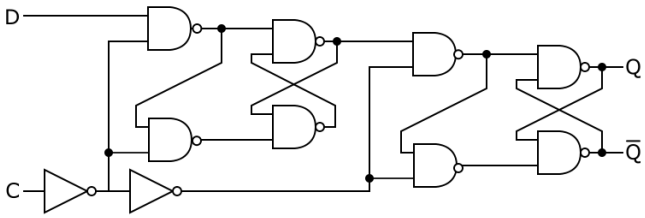
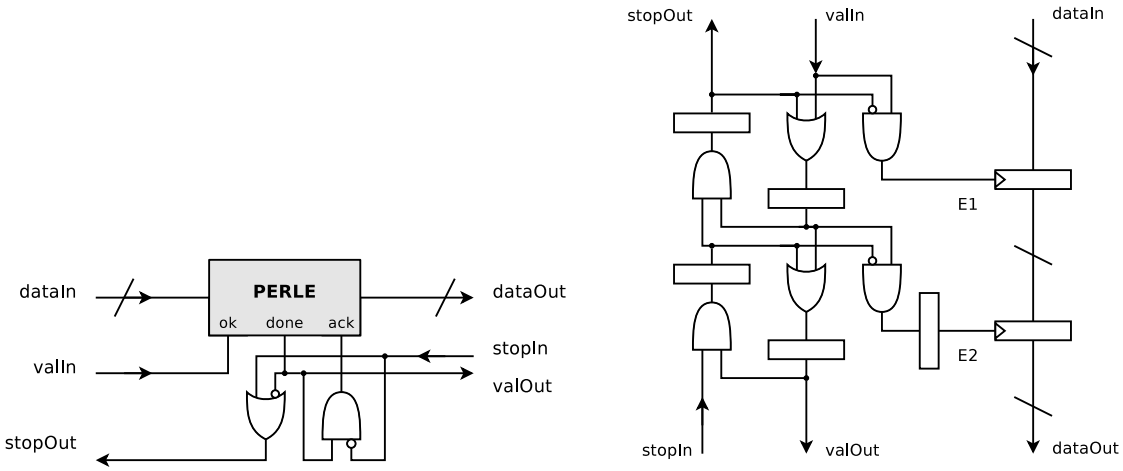


FIG. 6.10: Exemple de flip-flop sur front montant : bascules D en série, maître-esclave.



(a) Coquille. (b) Station relai.

FIG. 6.11: Implantation de Cortadella *et al.*, 2006.

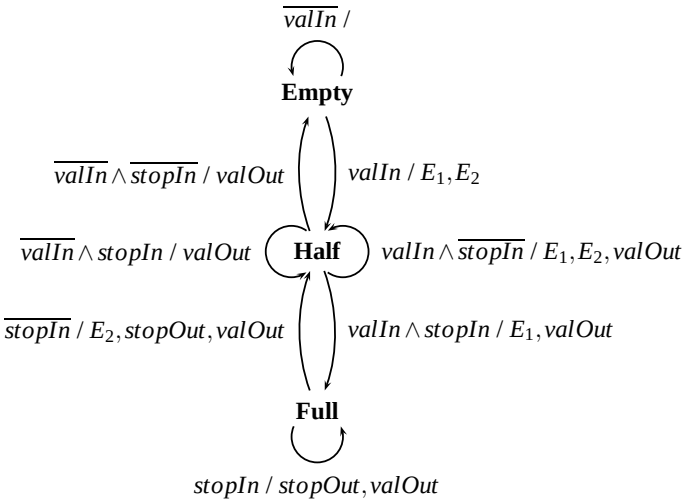


FIG. 6.12: Contrôle d'une station relai, Cortadella *et al.*, 2006.

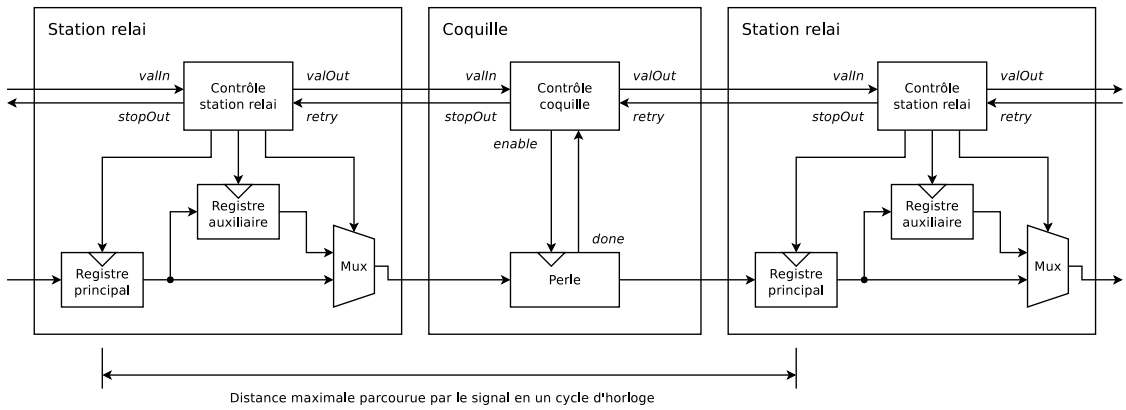


FIG. 6.13: Exemple de coquille avec une entrée et une sortie.

Cortadella *et al.* proposent donc de n'utiliser que $2n+5$ bascules transparentes par station relai, tout en jouant sur leurs horloges. Le verrouillage des bascules du flot de données dépend non plus de l'horloge, mais de la présence de données et de contre-pression. L'échantillonnage n'est plus systématique : la bascule se comporte comme un fil si elle reçoit une nouvelle donnée (*valIn*) et pas de contre-pression ($\overline{stopIn}$) ; elle est verrouillée sinon. Cette solution offre donc une réduction importante de la surface d'une station relai. Le détail de l'implantation est donnée en FIG. 6.11(b), tandis que l'automate correspondant est représenté en FIG. 6.12.

Alors que dans les implantations précédentes la perle est arrêtée par désactivation de son horloge, elle est ici supposée communiquer avec sa coquille par le biais de trois signaux : *go* commence le traitement, *done* marque la fin de calcul, et *ack* indique que la donnée a bien été reçue en aval. L'implantation est donnée en FIG. 6.11(a).

Notre variante

L'idée de base de cette nouvelle implantation consiste à retirer les registres des coquilles ; nous partons du principe que seules les stations relais ont vocation à contenir des données. Le rôle des coquilles est uniquement de rendre leurs perles patientes, conformément à la définition du LID. Nous proposons par conséquent une solution combinant le meilleur des implantations de Boucaron *et al.* de 2005 et de Cortadella *et al.* de 2006, sans ajout systématique de capacités de stockage. Les stations relais en amont transmettent directement leurs données à la perle, qui envoie directement ses résultats aux stations relais en aval (*cf.* FIG. 6.13).

Dans les implantations précédentes, la coquille dispose de registres à ses entrées pour stocker les données émises par les stations relais en amont. Puisque nous supprimons ces registres, nous devons modifier le protocole pour que la station relai maintienne l'émission de la donnée jusqu'à sa consommation. Son automate est reproduit en FIG. 6.14.

- La station relai reste dans l'état *Empty* tant qu'aucune donnée ne se présente en entrée. Lorsque *valIn* est présent, la donnée est stockée dans le registre principal, et l'automate passe en état *Retry*.
- Dans l'état *Retry*, la station relai contient une donnée. Elle conserve cet état si aucune autre donnée ne se présente tout en conservant la sienne, ou au contraire si elle transmet le contenu du registre principal tout en recevant une nouvelle valeur. Si la station relai reçoit à la fois de la contre-pression et une donnée, elle passe en état *Full*. Sinon, elle retourne en état *Empty*.

– Dans l'état *Full*, quels que soient les signaux en entrée, la station relai émet le contenu du registre auxiliaire et $stopOut$. Lorsque $retry$ est absent, la donnée a bien été reçue par le destinataire ; on retourne en état *Retry*.

De plus, les horloges des registres sont désactivées lorsque ceux-ci n'ont pas de nouvelles valeurs à échantillonner. Le comportement reste globalement inchangé par rapport à celui de Boucaron *et al.* 2005, à l'exception du fait qu'une valeur est répétée en sortie de la station relai jusqu'à ce qu'elle soit acquittée (absence de *retry*). La FIG. 6.15(a) présente la composition du contrôle de deux stations relais ; les signaux ne mettent pas plus d'un cycle d'horloge pour aller de l'une à l'autre. La logique d'une coquille est quant à elle illustrée en FIG. 6.15(b).

La perle est activée par le signal *enable*, et émet en retour le signal *done* pour indiquer que le calcul est terminé. Le signal *enable* est donc émis :

– si la coquille ne reçoit pas de contre-pression, donc que la station relai en aval a acquitté la précédente donnée, et que chaque station relai en entrée n'est pas vide, *ou*
– si la perle a déjà commencé un calcul sans l'avoir terminé.

En effet, la perle s'exécute en l'espace d'un *instant logique*, selon les hypothèses synchrones, et non d'un cycle d'*horloge physique*. Un exemple trivial est la multiplication, nécessitant plusieurs cycles d'horloge pour produire le résultat. Ainsi, pour tout i :

$$working = \left(\bigwedge_k \overline{retry_k}\right) \wedge \left(\bigwedge_k valIn_k\right) \wedge \overline{done} \tag{6.6}$$

$$enable = \left(\bigwedge_k \overline{retry_k}\right) \wedge \left(\bigwedge_k valIn_k\right) \vee \mathrm{pre}\,(working) \tag{6.7}$$

$$stopOut_i = \left(\bigvee_k retry_k\right) \vee valIn_i \wedge \left(\bigvee_{k\neq i} \overline{valIn_k}\right) \vee enable \wedge \overline{done} \tag{6.8}$$

$$valOut_i = done \tag{6.9}$$

Des expériences d'implantations sur FPGA ont été réalisées en utilisant le logiciel Xilinx ISE 10.1.02 sur les architectures Spartan3-1000 et Virtex5-LX50 (*cf.* TAB. 6.1 et 6.2). Pour chacune, nous avons testé les heuristiques d'optimisation de surface et de vitesse. Pour l'expérience de conception d'ASIC, nous avons utilisé FreePDK45 1.2 et la bibliothèque de cellules standard de l'Oklahoma State University [220, 221] avec Synopsys DC version Y-2006.06-SP4 (*cf.* TAB. 6.3). La fréquence d'horloge est de 1 GHz.

Pour chaque support, tables supérieures et inférieures se réfèrent respectivement à deux tests réalisés. Dans le premier cas, nous composons les parties de contrôle, pour un nombre croissant de stations relais connectées à une coquille. Dans le second cas, nous disposons d'un petit pipeline à deux entrées et une sortie. Nous comparons notre implantation avec celle de Boucaron *et al.*, 2005.

Concernant la surface de la partie contrôle d'une seule station relai pour ASIC, les résultats sont similaires avec un léger avantage à notre implantation ($61,0\ \mu m^2$ contre $64,8\ \mu m^2$, soit $-5,79\%$). En revanche, les résultats pour du FPGA ne sont pas assez significatifs pour pouvoir conclure.

Pour la composition des parties de contrôle, nous gagnons en surface et en vitesse aussi bien pour FPGA que pour ASIC. Avec un support FPGA, le gain en surface est compris entre -6% et -29%, tandis que le gain en fréquence d'horloge va de $+3\%$ à $+50\%$, selon les exemples et heuristiques considérés. Si la cible est un ASIC, à fréquence d'horloge imposée, le gain en surface est de l'ordre de -10%.

Enfin pour l'exemple du petit pipeline sur FPGA, le gain en surface est d'à peine -4% à -6% ;

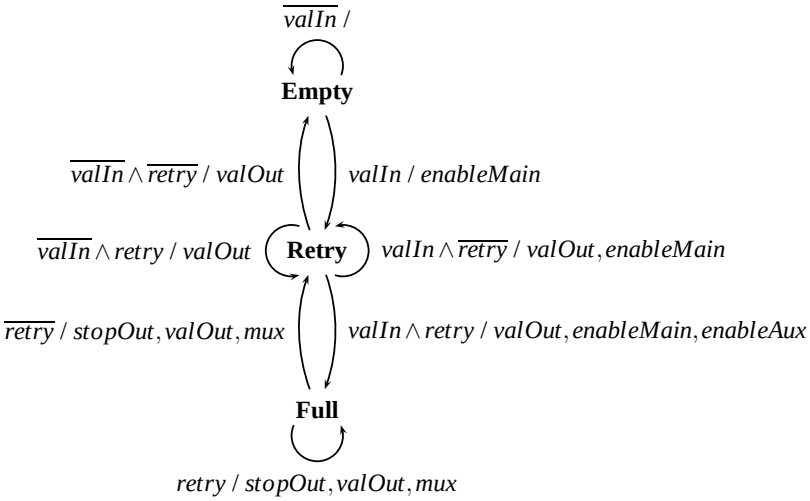
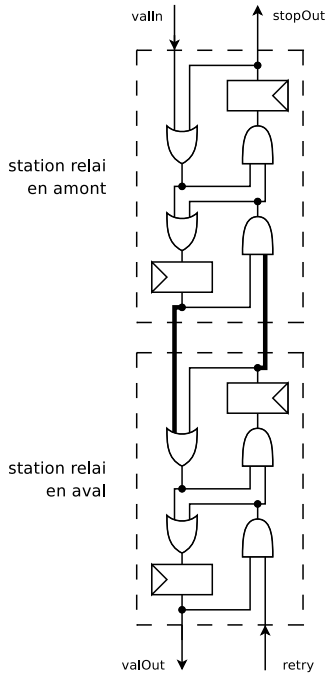
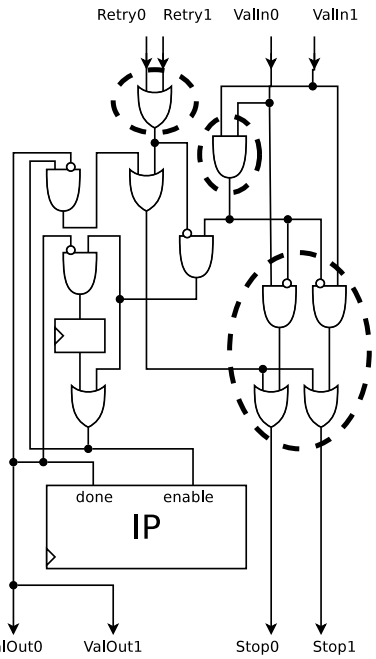


FIG. 6.14: Contrôle d'une station relai.



(a) Composition du contrôle de stations relais.



(b) Implantation d'une coquille à deux entrées et deux sorties. Les zones cerclées de pointillés indiquent les blocs de logique à modifier pour ajouter d'autres entrées/sorties.

FIG. 6.15: Logique de contrôle de stations relais (a) et d'une coquille (b).

In/Out	Contrôle, Boucaron *et al.* 2005				Contrôle, Boucaron *et al.* 2009				Gain			
	Optim. vitesse		Optim. surface		Optim. vitesse		Optim. surface		Optim. vitesse		Optim. surface	
	MHz	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*
2	240	24	237	20	246	17	250	17	+2.5%	-29%	+5%	-15%
4	169	50	136	39	246	17	250	17	+27%	-24%	+25%	-20%
8	148	87	114	81	167	70	146	64	+12%	-19%	+28%	-20%
16	128	153	129	157	182	120	182	120	+42%	-21%	+41%	-23%
32	110	311	117	312	166	235	167	235	+50%	-24%	+42%	-24%

In/Out	Multiplieur, Boucaron *et al.* 2005					Multiplieur, Boucaron *et al.* 2009					Gain			
	Optim. vitesse		Optim. surface			Optim. vitesse		Optim. surface			Optim. vitesse		Optim. surface	
	MHz	*LUT*	*MHz*	*LUT*	*FF*	*MHz*	*LUT*	*MHz*	*LUT*	*FF*	*MHz*	*LUT*	*MHz*	*LUT*
16x16	81.1	135	74.6	87	134	78.6	100	82.2	81	100	-3%	-26%	+10%	-6%
32x32	54.0	234	49.8	229	262	52.8	228	54.1	223	196	-2%	-2%	+8%	-2%

TAB. 6.1: FPGA – Spartan3-1000, speed -4

In/Out	Contrôle, Boucaron *et al.* 2005				Contrôle, Boucaron *et al.* 2009				Gain			
	Optim. vitesse		Optim. surface		Optim. vitesse		Optim. surface		Optim. vitesse		Optim. surface	
	MHz	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*	*MHz*	*LUT*
2	664	18	508	17	905	16	720	15	+36%	-11%	+41%	-11%
4	557	35	437	33	639	32	494	31	+14%	-8%	+13%	-6%
8	439	73	360	64	532	62	430	59	+21%	-15%	+19%	-7%
16	313	128	330	127	409	117	376	116	+30%	-8%	+13%	-8%
32	354	284	292	252	365	231	313	230	+3%	-18%	+7%	-8%

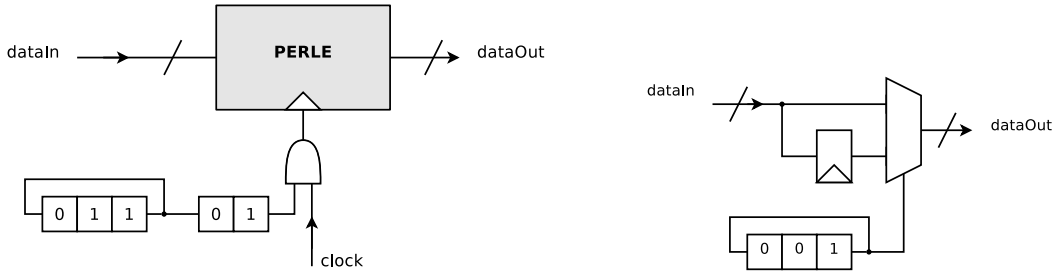
In/Out	Multiplieur, Boucaron *et al.* 2005					Multiplieur, Boucaron *et al.* 2009					Gain			
	Optim. vitesse		Optim. surface			Optim. vitesse		Optim. surface			Optim. vitesse		Optim. surface	
	MHz	*LUT*	*MHz*	*LUT*	*FF*	*MHz*	*LUT*	*MHz*	*LUT*	*FF*	*MHz*	*LUT*	*MHz*	*LUT*
16x16	192	84	190	82	134	229	78	229	78	100	+19%	-7%	+20%	-4%
32x32	105	148	104	146	262	116	142	116	142	196	+10%	-4%	+10%	-2%

TAB. 6.2: FPGA – Virtex5-LX50, speed -3

In/Out	Contrôle, Boucaron *et al.* 2005 Surface (μm^2)	Contrôle, Boucaron *et al.* 2009 Surface (μm^2)	Gain *(%)*
2	209	186	-10.7
4	424	378	-10.7
8	818	740	-9.5
16	1653	1489	-9.9
32	3352	2994	-8.9

In/Out	Multiplieur, Boucaron *et al.* 2005			Multiplieur, Boucaron *et al.* 2009			Gain		
	Surface	Puissance		Surface	Puissance		Surface	Puissance	
		dynamique	au repos		dynamique	au repos		dynamique	au repos
	(μm^2)	*(mW)*	*(μW)*	*(μm^2)*	*(mW)*	*(μW)*	*(%)*	*(%)*	(%)
16x16	5715	1.99	31.6	4791	1.22	26.0	-16.1	-38.7	-17.8
32x32	17986	3.70	92.1	16215	2.20	81.4	-9.80	-40.6	-11.6

TAB. 6.3: ASIC 45 nm, fréquence d'horloge 1 GHz



(a) Principe d'une coquille avec ordonnancement statique.

(b) Registre fractionnaire.

FIG. 6.16: Implantation de Casu et Macchiarulo, 2004.

les LUTs sont sous-exploitées par l'outil de placement et routage. En revanche, pour de l'ASIC, le gain est compris entre -10% et -16%.

Concernant la puissance, les résultats de l'implantation FPGA sont contradictoires, selon l'heuristique utilisée, donc non-significatifs. Pour une cible ASIC, nous avons un net gain en puissance dynamique de l'ordre de -40%, et entre -11% et -18% pour la puissance au repos.

6.1.2 Solutions à base d'ordonnancements statiques

En alternative aux protocoles dynamiques, les systèmes insensibles aux latences peuvent être ordonnancés statiquement. Certains travaux préconisent un ordonnancement à la compilation, évitant le surcoût de l'implantation du LIP [40, 56].

En effet, une implantation du LIP requiert des signaux supplémentaires tels que *val* ou *stop*, soit autant de fils à placer et router, et des registres auxiliaires parfois inutilisés. Si un ordonnancement statique du système est calculé au préalable, alors nous pouvons nous en passer : un ou plusieurs oracles contrôlent directement l'horloge de chaque élément. Une perle n'est activée que lorsqu'il est prévu qu'elle dispose de toutes ses données en entrée, et les conjestions sont évitées par anticipation.

Casu et Macchiarulo, 2004

Les travaux de Casu et Macchiarulo sont les premiers à proposer une implantation statique du LID [56]. Un ordonnancement périodique est tout d'abord calculé, tel que les données ne soient pas écrasées. Sur les canaux, les stations relais sont remplacées par de simples registres, distant les uns des autres d'au plus un cycle d'horloge.

Chaque coquille contient la séquence binaire ultimement périodique qui, à chaque cycle d'horloge, active ou non la perle. Cette séquence binaire est alors contenue dans un registre à décalage, jouant le rôle d'ordonnanceur. Ce principe est présenté en FIG. 6.16(a), même si l'implantation réelle est quelque peu différente[3]. Le registre de droite correspond à la phase d'initialisation, tandis que celui de gauche contient le régime périodique.

Comme nous l'avons vu précédemment avec les graphes marqués, le débit du système est borné par le débit de son circuit le plus lent (*cf.* sous-section 2.2). Si l'on retire la logique de contrôle et la contre-pression du système, alors il convient de retarder les circuits les plus rapides de telle façon qu'un nœud de calcul reçoive ses jetons en entrée *précisément* lorsqu'il doit les consommer. Casu et Macchiarulo nomment ce processus une *égalisation* : des flip-flops additionnelles sont insérées sur certains canaux pour ajouter des latences supplémentaires aux circuits trop rapides, et ainsi en

[3] Une bascule est nécessaire pour éviter les *glitchs* [56].

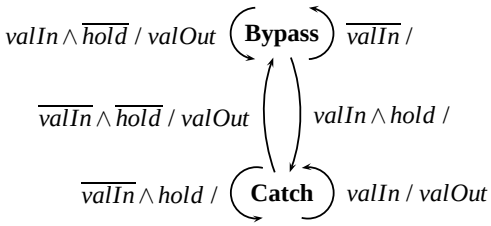


FIG. 6.17: Contrôle d'un registre fractionnaire, Boucaron *et al.*, 2006–2007.

réduire les débits. Nous renvoyons le lecteur à Boucaron *et al.* [40] pour une explication détaillée de l'égalisation et de l'ordonnancement statique du système.

Par ailleurs, il n'est pas toujours possible d'égaler le débit du circuit le plus lent en ajoutant des latences entières. Par exemple, considérons deux circuits, l'un de débit $2/3$, l'autre de débit $3/5$. Introduire une latence supplémentaire sur le premier circuit ferait chuter son débit à $2/4$, ralentissant ainsi l'ensemble du système. Dans ce cas précis, le nombre de latences à ajouter est rationnel. C'est pourquoi les auteurs introduisent alors le concept de *registre fractionnaire* : un registre muni d'un bypass, lui aussi ordonnancé statiquement (*cf.* FIG. 6.16(b)). Tantôt il retarde la propagation d'une donnée, tantôt il la laisse passer combinatoirement.

Le gain obtenu par suppression de la logique et des fils du LIP est donc à mettre en balance avec le coût induit par ces registres supplémentaires. Un autre inconvénient de la solution statique est qu'elle ne s'applique qu'à des systèmes où les latences de communication sont connues par avance, contrairement aux protocoles dynamiques.

Boucaron *et al.*, 2006–2007

L'inconvénient de l'implantation précédente du registre fractionnaire est qu'elle nécessite un important registre à décalage pour y stocker l'ordonnancement statique, alors que celui-ci peut être déduit des ordonnancements du producteur et du consommateur. Une telle solution fut proposée par Boucaron *et al.* [40] ; le registre à décalage est alors remplacé par un automate activant ou non l'échantillonnage du flot de données. La machine de Mealy correspondante est représentée en FIG. 6.17 :

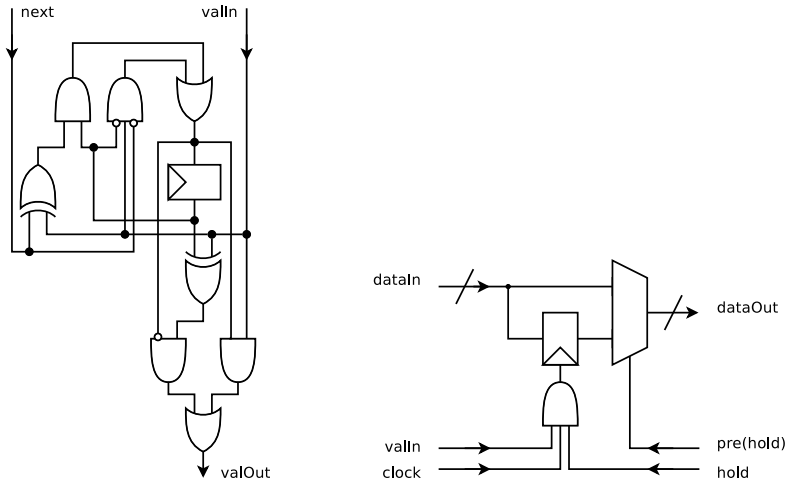
- Dans l'état *Bypass*, le flot de données n'est pas échantillonné, et le registre fractionnaire ne contient pas de données. L'état est conservé si le producteur n'émet pas de donnée (signal *valIn*), ou si la donnée reçue ne doit pas être stockée (signal *hold*). Sinon, le registre passe en état *Catch*.
- Le registre fractionnaire contient une donnée dans l'état *Catch*. S'il reçoit une donnée, celle déjà contenue dans le registre est émise et remplacée par la nouvelle. L'automate conserve aussi son état si la donnée stockée doit être retenue jusqu'au prochain instant. Sinon, elle est émise au consommateur, et l'automate retourne en état *Bypass*.

Le signal *hold* est donné par la formule :

$$\overline{valIn \oplus next} \wedge \mathrm{pre}\,(hold) \vee \overline{\mathrm{pre}\,(hold)} \wedge valIn \wedge \overline{next} \tag{6.10}$$

où *next* correspond à l'activité du consommateur. Le registre fractionnaire stocke une donnée si :

1. L'activité du consommateur égale celle du producteur $(valIn = next)$ et si le registre contient déjà une donnée, *ou*



(a) Contrôle. (b) Flot de données.

FIG. 6.18: Implantation d'un registre fractionnaire, Boucaron *et al.*, 2006–2007.

2. Si le registre ne contient pas de donnée, qu'il en reçoit une, et qu'elle n'est pas consommée dans l'instant.

L'implantation du registre fractionnaire est présentée en FIG. 6.18.

Notre implantation

Carmona *et al.* remarquent qu'une égalisation du système n'est pas nécessaire à son ordonnancement statique [53]. En effet, les travaux antérieurs partent du principe que les nœuds du système s'exécutent *au plus tôt* (ASAP), comme avec un protocole dynamique ; afin d'éviter des pertes de données, il est nécessaire de ralentir les jetons les plus rapides par insertion de latences supplémentaires. Pourtant, l'idée d'un ordonnancement statique est précisément d'imposer aux nœuds leurs instants d'exécution. Un système insensible aux latences, modélisé par un graphe marqué, est sans conflit (*cf.* théorème 2.9) ; nous sommes donc libres de choisir la politique d'ordonnancement à appliquer, et en particulier, nous pouvons opter pour un ordonnancement *au plus tard* (ALAP).

Un problème majeur de l'ordonnancement ASAP est l'accumulation de jetons. Comme étudié en sous-section 2.2, une composante connexe et non-fortement connexe peut s'exécuter à un débit égal à 1, à condition que ses mémoires soient suffisamment grandes pour ne pas le surcontraindre. Si une telle composante est reliée en sortie à une composante fortement connexe de débit strictement inférieur à 1, alors un ordonnancement ASAP provoque une accumulation entre les deux composantes. Ainsi, dans un système modélisé par un graphe marqué, tous les nœuds, y compris ceux des composantes non-fortement connexes, doivent s'exécuter au rythme du circuit le plus lent. Pour s'en convaincre, se reporter aux théorèmes 2.7 et 2.8. Par ailleurs, une solution évidente des équations de balances d'un graphe marqué est $\vec{1}$.

Par conséquent, nous remarquons qu'une solution consiste à utiliser conjointement les ordonnancements ASAP et ALAP :

– Les circuits critiques du graphe sont les plus lents ; ils doivent s'exécuter en ASAP. En régime périodique, dès qu'un jeton du circuit critique arrive en entrée d'un nœud, ce dernier *doit* pouvoir s'exécuter. Autrement dit, les jetons des circuits non-critiques passant par ce nœud doivent

Entrées : un graphe marqué temporisé avec capacités $g = \langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, L, K \rangle$.
Sorties : les ordonnancements et clusters des nœuds, et les capacités des places.

Étape 1 : réduire la complexité du système

1: appliquer des techniques de clusterisation, telles que proposées par Carmona *et al.* [53]

Étape 2 : énumérer les circuits et calculer leurs débits

2: rechercher l'ensemble de ses circuits par l'algorithme de Johnson [123]

3: identifier les circuits critiques

Étape 3 : calculer les ordonnancements des circuits critiques

4: calculer les ordonnancements des nœuds des circuits critiques par ordonnancement ASAP [40]

5: factoriser ces ordonnancements : réduire au plus les parties initiales par rotation des périodiques

Étape 4 : calculer les ordonnancements des circuits non-critiques

6: calculer les ordonnancements ALAP des autres nœuds, avec les cycles critiques pour références

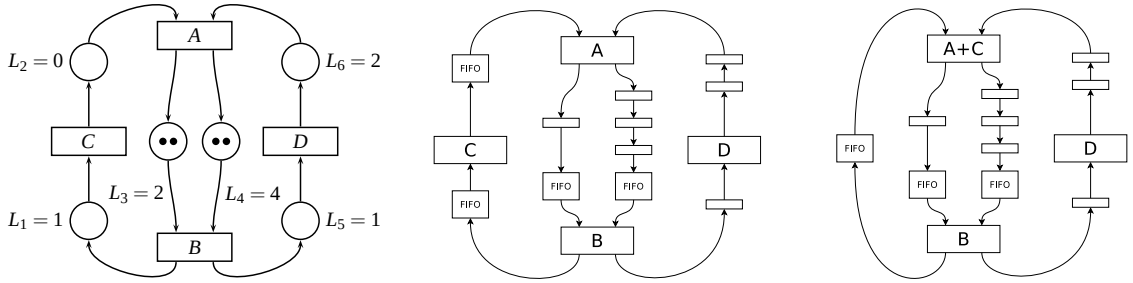
Étape 5 : dimensionner les mémoires de chaque canal

7: résoudre le programme linéaire en nombre entier de l'équation (2.16), ou des programmes linéaires de la forme de Lu-Koh [156], où les poids sur les arcs correspondent aux marquages initiaux

Étape 6 : s'assurer de la correction

8: le surcoût de la logique d'ordonnancement peut fausser les latences et les résultats précédemment calculés ; si tel est le cas, appliquer une technique de *recyclage* [51] et itérer depuis l'étape 2

Algorithme 6.1: Calcul de l'ordonnancement statique d'un LIS sans routage.



(a) Modélisation par un graphe marqué temporisé.

(b) Schéma fonctionnel du circuit correspondant, considérant C de latence non-nulle.

(c) Schéma fonctionnel du circuit correspondant, considérant C de latence nulle.

FIG. 6.19: Exemple de LIS avec notre implantation : une place correspond à une série de registres, suivie éventuellement d'une FIFO.

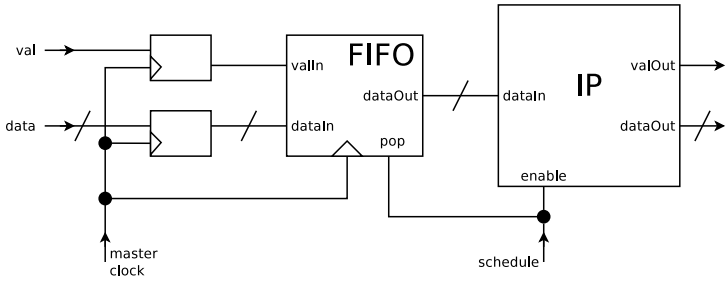


FIG. 6.20: Exemple d'implantation de canal, et lien avec l'IP.

arriver *avant* ou *au même instant*, mais jamais *après*, pour ne pas ralentir davantage le circuit critique.
- Les autres nœuds suivent un ordonnancement ALAP, avec pour référence les ordonnancements des circuits critiques. L'avantage de l'ordonnancement ALAP est de respecter naturellement les équations de balance ; il ne provoque pas d'accumulation de jetons, puisqu'ils ne sont produits que lorsque nécessaires.

Nous proposons l'algorithme 6.1 pour ordonnancer statiquement le système. Contrairement à l'algorithme de Carmona *et al.* [53], il n'est plus nécessaire de construire le graphe complémenté. D'autre part, nous clusterisons le graphe pour en réduire la complexité (étape 1) *avant* de l'ordonnancer (étapes 3 et 4) et d'appliquer le recyclage (étape 6) : en effet, la clusterisation a un impact sur la logique de contrôle et son surcoût de latences.

Aussi, les implantations précédentes demandent à ralentir les chemins les plus rapides, soit en introduisant des registres intermédiaires, soit en leur attribuant une horloge *ad hoc*, avec le surcoût pour la génération ou le stockage de la séquence k-periodique que cela implique. Des registres fractionnaires sont nécessaires pour égaliser les débits, lorsqu'il n'est pas possible de jouer sur des latences entières. Notre nouvelle implantation évite ces surcoûts.

Exemple 6.1. Considérons le graphe marqué de la FIG. 6.19(a). Les latences d'interconnexion sont indiquées à côté des places. Supposons des temps de calcul non-nuls, tels que le circuit de droite soit critique.

La FIG. 6.19(b) présente le schéma fonctionnel du circuit correspondant. Dans le circuit de droite, hormis la place qui contient initialement deux données, les autres sont traduites en de simples registres (flip-flops). Dans le circuit de gauche, les places correspondent à des registres et des FIFOs. S'il avait fallu égaliser les circuits, nous aurions ajouté des registres sur les canaux de latence L_1 et L_2 pour faire chuter le débit du circuit. À la place, notre implantation consiste à faire avancer les jetons jusqu'à la FIFO en entrée du nœud de calcul, et les consommer lorsque le nœud reçoit un jeton sur son autre entrée.

Supposons maintenant que le nœud C est un bloc de combinatoire de latence nulle. Il peut alors être fusionné avec le nœud A (étape 1 de l'algorithme 6.1) afin d'économiser le coût d'une FIFO, comme représenté en FIG. 6.19(c).

Le détail de l'implantation est représenté en FIG. 6.20 ; il correspond, dans l'exemple précédent, au canal entre A et B de latence L_3. Le registre correspond à n flip-flops en parallèle pour stocker une donnée de n bits, plus une flip-flop correspondant à l'état de la donnée (présente/absente). Le registre échantillonne son entrée à chaque cycle de l'horloge principale, et répète le résultat l'instant suivant ; un jeton est donc propagé *au plus tôt* à travers une série de registres. Lorsque ce jeton atteint une FIFO, il y est stocké. La tête de la file est retirée dès lors que l'IP (la perle) débute un nouveau calcul.

Ainsi, contrairement aux implantations précédentes, l'implantation est triviale. Il n'est pas nécessaire de stocker l'ordonnancement des registres et FIFOs comme dans l'implantation de Casu-Macchiarulo ; et contrairement à Boucaron *et al.*, elle ne nécessite aucune logique de contrôle additionnelle, hormis celui de la FIFO : *push* sur donnée entrante, *pop* sur activation de l'IP. Par ailleurs, comme nous avons vu que les nœuds d'un circuit ont le même ordonnancement périodique *modulo rotation*, celui-ci peut être distribué sur l'ensemble du circuit ; les décalages sont réalisés par ajout de latences sur l'arbre d'horloge, par exemple en insérant des bascules.

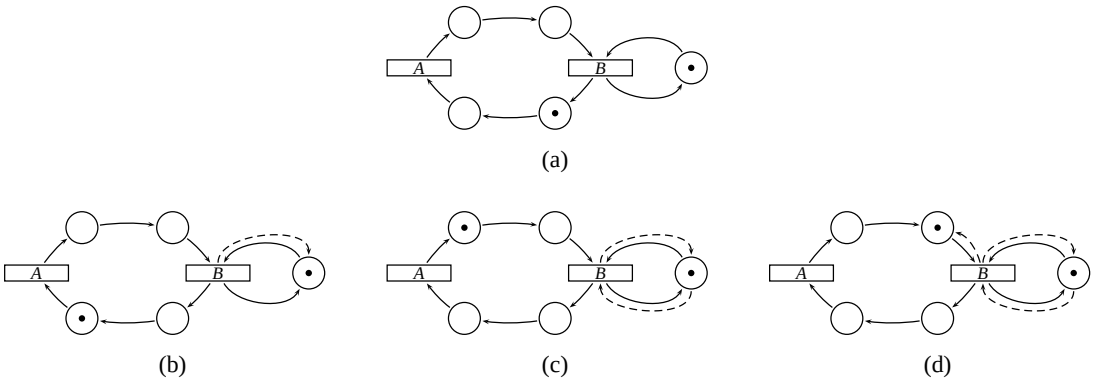


FIG. 6.21: Exemple de contre-pression entraînant un verrou mortel.

6.1.3 Erreurs récurrentes

Certaines des implantations étudiées jusqu'ici sont en fait subtilement incorrectes : le comportement du système insensible aux latences ainsi généré contredit les hypothèses du protocole. Étudions plus en détail ces erreurs à éviter.

Propagation de contre-pression non-nécessaire

La contre-pression ne doit être propagée que lorsque celle-ci est strictement nécessaire à la préservation de l'intégrité des données : une station relai ne doit émettre un signal d'arrêt vers l'amont que lorsque ses deux registres sont pleins et qu'elle ne peut recevoir de nouvelles données. De même, une coquille ne doit en émettre que lorsqu'au moins une de ses sorties est saturée, et qu'une ou plusieurs données produites peuvent être perdues. Dans le cas contraire, si la contre-pression est systématiquement propagée, le débit du système n'est pas optimal. En particulier dans un système cyclique, un encombrement ponctuel ne peut se résorber, et de la contre-pression «parasite» continue à se propager dans le circuit alors qu'elle n'est plus nécessaire. Au pire, cela mène à une situation de verrou mortel.

Un exemple est donné en FIG. 6.21(a). Le système est composé de deux processus patients *A* et *B* (coquilles et perles), représentés par des rectangles. Ces processus sont reliés entre eux par des fils munis de stations relais, représentées par les cercles. Dans la situation initiale, deux stations relais contiennent chacune une donnée ; *B* a besoin de la donnée du circuit gauche pour pouvoir s'exécuter.

1. Au premier instant (*cf.* FIG. 6.21(b)), le jeton du circuit de gauche est transféré à la station relai suivante. *B* ne peut consommer la donnée disponible sur son autre entrée, et émet de la contre-pression.
2. Si la station relai émet *stopOut* dès qu'elle reçoit *stopIn*, alors à l'instant suivant, elle émet à son tour de la contre-pression vers *B*, bien qu'elle soit à moitié vide (*cf.* FIG. 6.21(c)).
3. *B* est bloqué par la contre-pression de l'une de ses sorties, alors qu'il en est lui-même à l'origine. *B* envoie des signaux d'arrêt à l'ensemble de ses entrées, menant progressivement à un blocage complet du système (*cf.* FIG. 6.21(d)).

L'implantation de Carloni *et al.* (*cf.* sous-section 6.1.1) peut mener à ce genre de scénario. En effet, la contre-pression y est calculée selon l'équation :

$$stopOut_i = \left(\bigvee_k voidIn_k \right) \vee \left(\bigvee_k stopIn_k \right) \tag{6.11}$$

Permutations de données

Les implantations de Carloni *et al.* 1999, Casu-Macchiarulo 2003, et Carloni 2006 ont en commun d'introduire des permutations sur les flots de données (*cf.* sous-section 6.1.1).

Les registres principaux et auxiliaires des stations relais sont en parallèle, comme représentés en FIG. 6.2(b) et 6.8(b). D'après leurs automates de contrôle (*cf.* FIG. 6.3, 6.4 et 6.9), les stations relais écrivent, puis lisent, dans leurs registres auxiliaires, avant de retourner en régime normal. Considérons deux jetons successifs ① et ②, et trois situations différentes :

1. Dans le premier cas, les arrivées des jetons ① et ② sont espacées dans le temps, et aucune contre-pression n'intervient. ① entre puis quitte la station relai, de même pour ②. Les ordres des jetons en entrée et en sortie sont identiques : (①, ②).
2. Supposons maintenant qu'une congestion survienne en aval. ① est stocké dans le registre principal et ② dans l'auxiliaire. Lorsque la congestion se résorbe, le premier jeton émis est celui du registre auxiliaire, ②, avant d'émettre ①. L'ordre en entrée est (①, ②), tandis que l'ordre en sortie est (②, ①).
3. Cependant, en retardant suffisament l'arrivée de ② dans le précédent scenario, ① aurait pu sortir en premier de la station relai, et il n'y aurait pas de permutation.

Ainsi, des permutations de ① et ② peuvent se produire, à cause de leurs instants d'arrivée et non de leurs valeurs. Cela contredit la définition d'un processus patient ; ces implantations ne vérifient pas les hypothèses du protocole.

6.2 Architecture d'un flot de données avec routage

Agival, Singh et Theobald ont proposé d'étendre le cadre du LID en permettant de dupliquer des nœuds de calcul, et de multiplexer et démultiplexer les envois de valeurs vers leurs diverses occurrences, afin d'augmenter la capacité de traitement du système [3, 215]. Dans le cadre de routages k-périodiques, cette extension du LID peut être modélisée KRG : les nœuds de fusion et de sélection font office, dans l'idée d'Agival *et al.*, de multiplexeurs et démultiplexeurs.

Un certain nombre d'autres travaux ont par la suite proposé un raffinement de cette idée, sous le nom de protocoles insensibles aux latences adaptatifs (*Adaptative Latency-Insensitive Protocols* – ALIP) [57, 58, 69, 150]. Ils reposent sur l'un des deux concepts suivant :

1. Lorsqu'une perle ne consomme qu'une partie de ses entrées, un *antijeton* est rétropropagé sur les autres entrées. Lorsqu'un antijeton rencontre une donnée, ils s'annihilent. Le coût de cette solution est un doublement du protocole LIP : le réseau dual propage les antijetons.
2. Les coquilles disposent de compteurs sur chacune de leurs entrées. Lorsque la perle s'exécute sans consommer de donnée sur l'une de ses entrées, le compteur correspondant est incrémenté. D'autre part, lorsqu'une donnée se présente sur une entrée dont la valeur du compteur est non-nulle, alors la donnée est détruite et le compteur décrémenté.

Ces protocoles supportent une large expressivité du modèle sous-jacent –les productions et consommations de jetons peuvent dépendre des données–, mais limitent d'autant les optimisations possibles à la compilation. Ils permettent notamment de concevoir des pipelines avec spéculation [90]. Enfin, ces protocoles autorisent la destruction de jetons, ce que nous nous sommes interdit jusqu'à présent.

Dans cette sous-section, nous nous proposons donc de développer une alternative à ces implantations, en collant à la sémantique des KRG.

6.2.1 Avec protocole adaptatif

Nous présentons notre protocole adaptatif. Il consiste en une extension de notre protocole préalablement introduit en sous-section 6.1.1. Les stations relais et coquilles restent inchangées par rapport aux FIG. 6.14, et 6.15. Puis nous présentons l'implantation de l'interconnexion et du routage : nœuds de sélection, de fusion et de copie.

Coquille

La coquille fait office d'interface entre sa perle et son environnement. Ainsi, la coquille active sa perle par le signal $Enable$, qui assure son *clock gating*. Lorsque $Enable$ est présent, la perle traite ses données en entrée ; lorsqu'il est absent, la perle est désactivée. C'est donc à la coquille de s'assurer que le calcul peut être effectué et transmis vers l'aval. La perle retourne un signal *Done* lorsque le calcul est terminé. Le registre *Reg* conserve l'information, d'un cycle d'horloge à l'autre, qu'un calcul a été entrepris et non terminé.

D'autre part, l'environnement envoie à la coquille des signaux *ValIn* lorsque des données sont disponibles sur ses entrées, et des signaux *Retry* en cas de congestion en aval. Lorsqu'un calcul est terminé, la coquille émet *ValOut* vers l'aval. Au contraire, elle émet *Stop* vers l'amont si elle reçoit de la contre-pression ou si elle ne peut s'exécuter faute de données en entrée. À tout instant i, le contrôle d'une coquille à m entrées et n sorties est donné par les équations suivantes :

$$Enable_i = Fireable_i \vee Reg_i \tag{6.12}$$
$$Fireable_i = \overline{Retry_i} \wedge Present_i \tag{6.13}$$
$$Present_i = \bigwedge_{k=0}^{m-1} ValIn_i(k) \tag{6.14}$$
$$Reg_1 = 0 \tag{6.15}$$
$$Reg_{i+1} = Fireable_i \wedge \overline{Done_i} \tag{6.16}$$
$$Retry_i = \bigvee_{k=0}^{n-1} Retry_i(k) \tag{6.17}$$
$$Stop_i = Retry_i \vee \left(Enable_i \wedge \overline{Done_i}\right) \tag{6.18}$$
$$Stop_i(k) = Stop_i \vee ValIn_i(k) \wedge \overline{Present_i} \tag{6.19}$$
$$\forall k,\ ValOut_i(k) = Done_i \tag{6.20}$$

Station relai

Nous avons déjà étudié le contrôle de la station relai en FIG. 6.14. Nous donnons ci-dessous son code Esterel. Les deux registres déterminent dans lequel des trois états se situe l'automate :

1. $\overline{Reg0} \wedge \overline{Reg1}$: état $Empty$
2. $\overline{Reg0} \wedge Reg1$: état $Retry$
3. $Reg0 \wedge Reg1$: état $Full$

Notons que l'état $Reg_0 \wedge \overline{Reg_1}$ est inatteignable : le registre principal ne peut être vide lorsque le registre auxiliaire contient une donnée. Les signaux *EnableMain* et *EnableAux* activent les registres correspondants de la station relai ; les registres n'échantillonnent leurs entrées que lorsque ces signaux

sont présents. Enfin, le signal *Mux* dirige le multiplexeur : s'il est présent, la donnée émise est celle du registre *Aux*, sinon celle du registre *Main*.

$$
\begin{aligned}
Tmp0_i &= ValIn_i \vee Reg1_i && (6.21)\\
Tmp1_i &= Retry_i \wedge Reg0_i && (6.22)\\
Reg0_1 &= 0 && (6.23)\\
Reg1_1 &= 0 && (6.24)\\
Reg0_{i+1} &= Tmp0_i \vee Tmp1_i && (6.25)\\
Reg1_{i+1} &= Tmp0_i \wedge Tmp1_i && (6.26)\\
EnableAux_i &= EnableMain_i \wedge Reg1_i \wedge Retry_i && (6.27)\\
EnableMain_i &= ValIn_i \wedge \overline{Reg0_i} && (6.28)\\
Mux_i &= Reg0_i \wedge Reg1_i && (6.29)\\
Stop_i &= Reg0_i && (6.30)\\
ValOut_i &= Reg1_i && (6.31)
\end{aligned}
$$

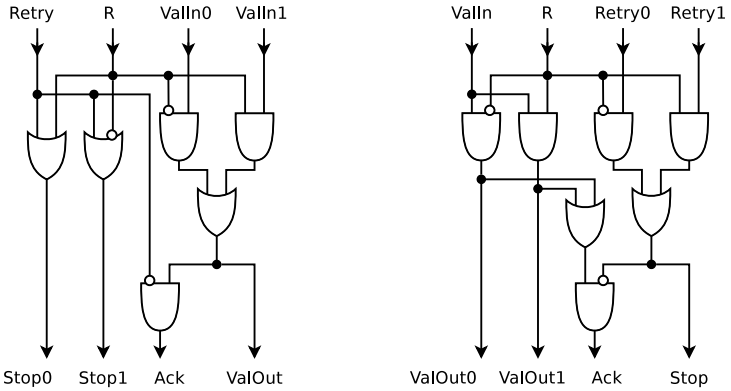
Fusion et sélection

Nous abordons maintenant l'implantation des nœuds d'interconnexion propres aux KRGs. Tout d'abord, le nœud de fusion. Il reçoit de l'amont des données signalées par $ValIn_0$ et $ValIn_1$, et rétropropage la contre-pression par les signaux $Stop_0$ et $Stop_1$. Une donnée est routée en sortie par l'émission de *ValOut*, tandis que la contre-pression en aval est reçue sur le port *Retry*. Les signaux *R* et *Ack* assurent la communication entre le nœud de fusion et l'oracle produisant la séquence de routage. Le signal *R* est égal à la valeur de la tête de ce registre. Lorsqu'une donnée est routée par le nœud de fusion, le signal *Ack* indique à l'oracle que la tête de la séquence est consommée.

$$
\begin{aligned}
Stop0_i &= Retry_i \vee R_i && (6.32)\\
Stop1_i &= Retry_i \vee \overline{R_i} && (6.33)\\
ValOut_i &= ValIn0_i \wedge \overline{R_i} \vee ValIn1_i \wedge R_i && (6.34)\\
Ack_i &= ValOut_i \wedge \overline{Retry_i} && (6.35)
\end{aligned}
$$

Nous ne précisons pas l'implantation de l'oracle, et laissons au développeur le choix de la solution à adopter, du moment qu'elle respecte le protocole R/Ack. En effet, une implantation basique consiste à utiliser un registre à décalage. Cette solution a l'avantage d'être simple, mais coûteuse en surface ; la séquence de routage pouvant être longue. Selon la forme de cette séquence, il peut être plus avantageux d'utiliser une méthode de compression (*e.g.* codage par plages), et/ou un automate la générant dynamiquement. Ce problème doit être résolu au cas par cas.

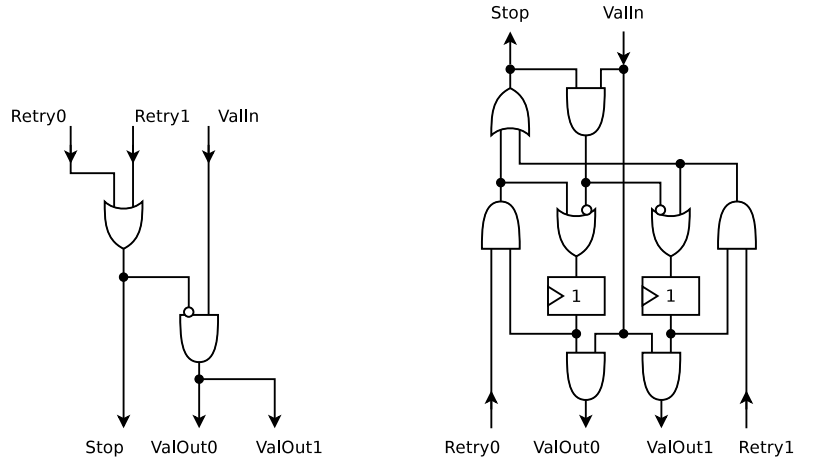
Très similaire à la fusion, la sélection route une donnée sur sa sortie indiquée par le signal *R*, et rétropropage le signal *Stop* si cette même sortie est congestionnée. Les logiques de contrôle des nœuds de fusion et de sélection sont illustrées en FIG. 6.22.



(a) Logique de la fusion.

(b) Logique de la sélection.

FIG. 6.22: Implantation du contrôle des nœuds de routage.



(a) Logique d'une copie passive 1 vers 2.

(b) Logique d'une copie urgente 1 vers 2.

FIG. 6.23: Implantations du contrôle des copies.

$$Stop_i = \left(Retry0_i \wedge \overline{R_i}\right) \vee \left(Retry1_i \wedge R_i\right) \quad (6.36)$$

$$ValOut0_i = ValIn_i \wedge \overline{R_i} \quad (6.37)$$

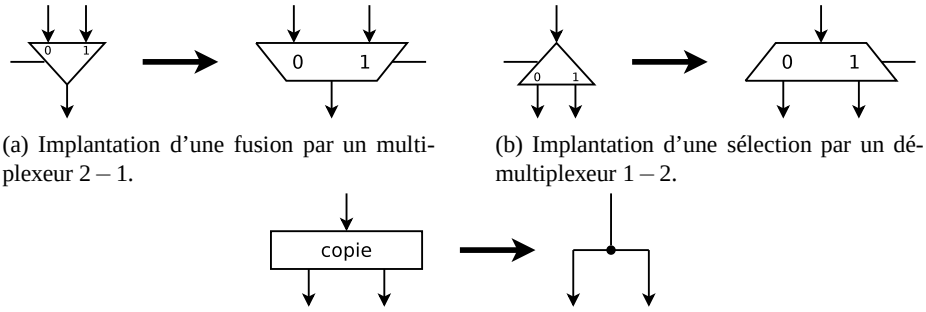
$$ValOut1_i = ValIn_i \wedge R_i \quad (6.38)$$

$$Ack_i = \left(ValOut0_i \vee ValOut1_i\right) \wedge \overline{Stop_i} \quad (6.39)$$

Copie

Enfin, nous proposons deux implantations de la copie : la copie *paresseuse* et la copie *urgente*. Elles sont toutes deux équivalentes aux *lazy fork* et *eager fork*, respectivement, de Cortadella *et al.* [70]. Les logiques de contrôle des deux implantations des nœuds de copie sont représentées en FIG. 6.23.

Dans le cas d'une copie paresseuse, la donnée en entrée n'est transmise aux sorties que lorsque chacune d'elles est apte à la recevoir. Si l'une des sorties remonte de la contre-pression, c'est l'en-



(a) Implantation d'une fusion par un multiplexeur $2-1$.

(b) Implantation d'une sélection par un démultiplexeur $1-2$.

(c) Implantation d'une copie.

FIG. 6.24: Implantation statique du routage d'un KRG.

semble des copies qui sont arrêtées. Les équations suivantes correspondent au contrôle d'une copie «1 vers n».

$$Stop_i = \bigvee_{k=0}^{n-1} Retry_i(k) \tag{6.40}$$

$$\forall k,\ ValOut_i(k) = ValIn_i \wedge \overline{Stop_i} \tag{6.41}$$

Bien que plus complexe, la copie urgente permet d'améliorer les performances si les sorties ont des contre-pressions différentes. Cette copie envoie la donnée indépendamment sur chacune de ses sorties, jusqu'à ce qu'elle soit transmise à l'ensemble des destinataires. Des registres sont nécessaires sur chaque sortie pour mémoriser quels destinataires ont déjà reçu la donnée. Les équations suivantes correspondent également au contrôle d'une copie «1 vers n».

$$\forall k,\ Reg_1(k) = 1 \tag{6.42}$$

$$\forall k,\ Reg_{i+1}(k) = Reset_i \vee Resend_i(k) \tag{6.43}$$

$$\forall k,\ Resend_i(k) = Retry_i(k) \wedge Reg_i(k) \tag{6.44}$$

$$Reset_i = \overline{ValIn_i \wedge Stop_i} \tag{6.45}$$

$$Stop_i = \bigvee_{k=0}^{n-1} Resend_i(k) \tag{6.46}$$

$$\forall k,\ ValOut_i(k) = ValIn_i \wedge Reg_i(k) \tag{6.47}$$

6.2.2 Avec ordonnancement statique

L'implantation de systèmes basés sur les KRG, avec ordonnancement statique, présente des similitudes avec celle des graphes marqués, présentée en sous-section 6.1.2. Les deux différences majeures reposent d'une part sur une interconnexion plus complexe, et d'autre part sur le calcul de l'ordonnancement.

Interconnexion

Les KRG ont trois nouveaux éléments par rapport aux graphes marqués, à savoir les fusions, les sélections et les copies. Ici, les fusions et les sélections, et par extension les n-fusions et les n-

Entrées : un KRGS $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M, R, K, L, T \rangle$.
Sorties : les ordonnancements et clusters des nœuds, et les capacités des places.

Étape 1 : réduire la complexité du système

1: appliquer des techniques de clusterisation, telles que proposées par Carmona *et al.* [53]

Étape 2 : calculer les ordonnancements ASAP

2: simuler une exécution ASAP du KRGS, sous les contraintes du théorème 3.69, et en déduire les ordonnancements des nœuds et places

3: factoriser ces ordonnancements : réduire au plus les parties initiales par rotation des périodiques

Étape 3 : réordonnancement des nœuds non critiques

4: pour tout jeton qui n'est pas consommé immédiatement par un nœud, retarder son instant de production, à moins que cela retarde un autre consommateur (nous cherchons des ordonnancements ALAP).

Étape 4 : s'assurer de la correction

5: le surcoût de la logique d'ordonnancement peut fausser les latences et les résultats précédemment calculés ; si tel est le cas, appliquer une technique de *recyclage* [51] et itérer depuis l'étape 2

Étape 5 : calculer les capacités des places

6: À partir des ordonnancements et des résultats de la sous-section 3.5.3, calculer les capacités des places.

Algorithme 6.2: Calcul de l'ordonnancement statique d'un LIS avec routage.

sélections, sont de simples multiplexeurs et démultiplexeurs (*cf.* FIG. 6.24(a) et 6.24(b)). Les copies, quant à elles, ne sont rien de plus que des embranchements entre différents chemins (*cf.* FIG. 6.24(c).

Ordonnancement

Pour l'implantation statique d'un LIS sans routage, nous avons proposé l'algorithme 6.1 afin d'en calculer l'ordonnancement. Ici, cet algorithme ne peut s'appliquer directement : nous ne pouvons pas énumérer les circuits critiques d'un KRG, pour la simple raison que ce concept n'est pas défini dans un KRG. En raison du routage variable d'un KRG, les circuits dans le graphe évoluent au cours du temps ; en général, ils ne sont pas fixes comme dans un graphe marqué.

Par contre, nous avons abordé le problème de l'ordonnancement des KRG en sous-section 3.5.2, et avons vu qu'il existe d'autres critères adaptés au KRG. Le théorème 3.69 impose un ensemble de contraintes sur les ordonnancements, pour que ceux-ci soient valides. Nous pouvons donc calculer un ordonnancement du KRG par simulation, de façon à respecter les contraintes de *synchronisabilité* des horloges des producteurs et consommateurs, et de *précédence* ou *synchronisation* de leurs instants d'exécution. L'ordonnancement d'un KRG s'écrit alors simplement sous la forme de l'algorithme 6.2.

Quatrième partie

Conclusions et annexes

Chapitre 7

Conclusion

> « Quarante-deux ! cria Loonquawl. Et c'est tout ce que t'as à nous montrer au bout de sept millions et demi d'années de boulot ?
> — J'ai vérifié très soigneusement, dit l'ordinateur, et c'est incontestablement la réponse exacte. Je crois que le problème, pour être tout à fait franc avec vous, est que vous n'avez jamais vraiment bien saisi la question. »
>
> Douglas Adams, *The Hitchhiker's Guide to the Galaxy*, 1979.

Nous avons étudié au cours des chapitres la sémantique et les propriétés des graphes à routage k-périodique, leurs relations avec certains modèles de l'état de l'art, ainsi que leur positionnement entre les modèles polyédriques et synchrones. Pourtant, différents points restent en suspend, et des travaux complémentaires doivent être conduits afin de les intégrer dans un flot de compilation performant. Dans les sections suivantes, nous proposons et discutons différentes idées visant à améliorer ces résultats théoriques.

7.1 Contrôle paramétré

Le modèle polyédrique permet de modéliser des applications paramétrées. Nous avons fait la supposition que les intervalles de valeurs de ces paramètres étaient connus à la compilation, mais pas leurs valeurs exactes, qui ne sont *a priori* spécifiées qu'à l'exécution.

En son état actuel, le routage des KRG est statique ; il doit être calculé dès la construction du graphe à partir du modèle polyédrique. De ce fait, nous avons imposé au chapitre 5 d'expliciter les valeurs des paramètres, pour construire le KRG correspondant. L'inconvénient que présente cette solution est évidemment de perdre en généricité : un KRG est construit pour un jeu de paramètres donnés. Il serait donc intéressant d'envisager une extension du routage des KRG au cas paramétré.

De la même manière que nous avons fait le rapprochement entre KRG et CSDF, une solution consiste à appliquer le principe des flots de données cyclo-dynamiques aux KRG (*cf.* sous-section 2.5.2). Les séquences de routages seraient alors exprimées par des séquences binaires paramétrées (*e.g.* $R(n) = 1^i.(0^j.1^k)^\omega$) où les valeurs des paramètres (ici i, j et k) seraient portées par des jetons de contrôle, lors de l'exécution. Notons que la solution proposée par Sen *et al.* consiste en un métamodèle, HPDF, pouvant s'appliquer à divers réseaux de processus flot de données, dont SDF ou CSDF

[115, 211, 212]. Puisque nous avons établi des liens entre les KRG et ces deux derniers modèles, il s'agirait donc d'étudier dans quelle mesure HPDF peut également s'appliquer aux KRG. Et surtout, si l'expressivité qui en découle est suffisante pour effectuer le rapprochement entre modèle polyédrique paramétré et KRG paramétré.

Notons par ailleurs que l'ensemble des positions des «0» ou des «1» au sein d'une séquence k-périodique peut s'écrire sous la forme d'un ensemble d'entiers periodiques [161]. Par exemple, considérons la séquence suivante :

$$w = 01.(0110)^{\omega} \tag{7.1}$$

Les positions de ses «1» appartiennent à un ensemble d'entiers périodiques :

$$\{\mathrm{idx}_1(w,i) \mid i \geq 1\} = \{2\} \cup \{4+4i \mid i \geq 0\} \cup \{5+4i \mid i \geq 0\} \tag{7.2}$$

Nous pensons que cette représentation pourrait être une piste pour exprimer de façon algébrique le lien entre, d'une part, le routage des jetons dans le KRG et, d'autre part, les coordonnées des opérations correspondantes dans le modèle polyédrique. En effet, l'énumération des opérations n'est nécessaire à la construction des routages des n-fusions et des n-sélections que si ceux-ci doivent être explicites (*cf.* section 5.2). Si le routage était paramétré, nous pourrions en garder une représentation ensembliste.

7.2 Dualité KRG-GDR

Nous avons étudié, en section 3.2, la construction de graphes de dépendances à partir d'un KRG. Inversement, nous avons montré qu'une forme normale de KRG, le quasi-graphe marqué, est construit à partir d'un graphe de dépendance réduit. Nous pouvons donc passer d'une représentation à l'autre, tout en conservant la fonctionnalité du système. De façon similaire, Grün *et al.* [110] et Hanzálek [116] ont tous deux établi qu'un graphe marqué pouvait être modélisé au moyen d'un graphe de dépendance, et réciproquement.

Implicitement, cela sous-entend que les graphes de dépendance d'une part, et les KRG ou graphes marqués d'autre part, sont des représentations duales : le KRG ou le graphe marqué décrit les relations entre opérateurs, tandis que le graphe de dépendance reflète les relations entre opérandes. De façon imagée, nous pouvons faire le parallèle avec les domaines temporels et fréquentiels, en terme de dualité : on passe du temps à la fréquence par une transformée de Fourier, et l'on passe du graphe marqué au graphe de dépendance par analyse des dépendances. Dans les deux cas, c'est la même information qui est exprimée selon deux points de vue différents.

Aussi nous pensons qu'une formalisation de ce principe de dualité pourrait être intéressante, notamment pour revisiter la construction d'un KRG paramétré (*cf.* point précédent) à partir d'un graphe de dépendance polyédrique.

7.3 Liens entre les modèles polyédrique et synchrone

Nous pouvons légitimement nous poser la question de l'impact de cette dualité sur le modèle final de notre flot de conception, à savoir le modèle synchrone.

Notion de temps multidimensionnel

Le modèle polyédrique ne dispose pas d'un *modèle* de temps au sens que l'entend la théorie du synchrone : il ne représente que des relations entre données dans l'espace. Il n'en demeure pas moins

que ce modèle contient une *notion* de temps. C'est précisément cette notion de temps que l'ordonnancement d'un modèle polyédrique vise à extraire (*cf.* section 4.3). À la différence du modèle synchrone, le «temps» polyédrique est éventuellement multidimensionnel. Cela n'est pas problématique en soit, Feautrier ayant prouvé qu'à tout ordonnancement multidimensionnel correspond un ordonnancement unidimensionnel (*cf.* théorème 1 de [87]). Il serait néanmoins intéressant de pouvoir exploiter cette notion de temps multidimensionnel en l'état ; une linéarisation (projection) introduisant une perte d'information. Dans ce sens, un temps logique multidimensionnel, incarné par des horloges vectorielles, fut introduit indépendamment par Fidge et Mattern [89, 162]. Cette idée généralise à n dimensions le concept d'instants logiques, défini par Lamport une décénie plus tôt.

Aussi, l'ordonnancement du modèle polyédrique, que nous avons rappelé en section 4.3, peut-être compris de façon totalement différente s'il est *pensé en terme d'horloges*. Le fait d'ajouter aux polyèdres les dimensions de leurs ordonnancements, comme le propose Bastoul (*cf.* sous-section 4.3.3), revient à coller ensemble des horloges multidimensionnelles et l'espace du modèle polyédrique. Autrement dit, nous modélisons ainsi un *espace-temps*.

Une illustration d'espace-temps : le modèle de Minkowski

S'il est un domaine des sciences où la corrélation entre *espace* et *temps* est étudiée dans ses détails, c'est bien la théorie de la relativité. Suite à la définition de son modèle de temps, Mattern notait une analogie avec le modèle de Minkowski dans un cas particulier à deux dimensions[1] [162]. Ce n'est certes qu'une illustration, mais à travers elle nous devinons le lien fondamental qui unit le modèle polyédrique aux horloges multidimensionnelles.

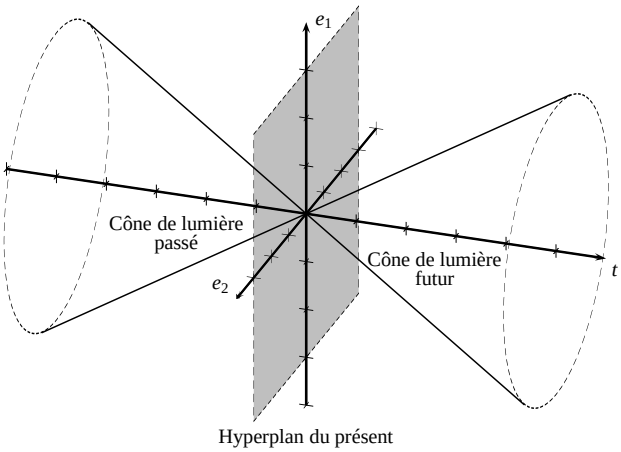
Rappelons que le modèle de Minkowski [170, 177], fondamental dans la théorie de la relativité restreinte, représente le continuum espace-temps. Il fut défini dans le but de «géométriser» la théorie de la relativité restreinte, et lui offrir un cadre mathématique proche de la géométrie euclidienne. C'est donc un cadre géométrique dans lequel sont étudiées des relations de *causalité* entre événements. L'*intervalle espace-temps* entre deux événements s'exprime par $c^2\Delta t^2 - \Delta d^2$, où Δt et Δd sont respectivement les distances temporelles et spatiales entre les événements. Deux événements peuvent dépendre l'un de l'autre si cet intervalle est positif. À la différence du modèle synchrone, il ne peut y avoir de dépendance dans l'instant.

Bien sûr, nos modèles ne manipulent que des dimensions *logiques*, et ne font intervenir ni la distance spatiale, ni la vitesse de la lumière. Mais de façon très simplifiée, nous calculons cet intervalle par l'équation (4.13) : la différence entre les coordonnées des instants doit être non-nulle ($\Delta t^2 > 0$) pour qu'il y ait dépendance, et plus précisément cette dépendance est *temporelle future* ($\Delta t > 0$) afin de garantir la causalité.

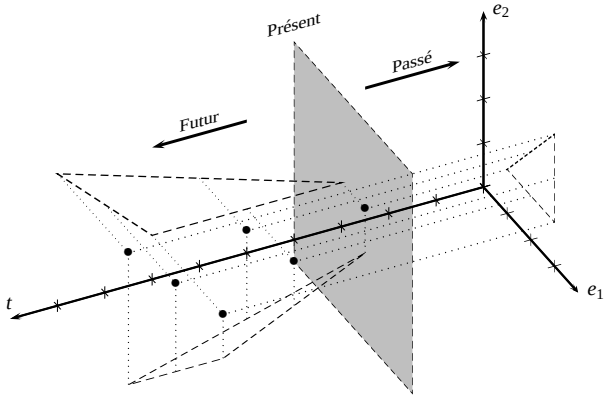
La FIG. 7.1 met en perspective les analyses de dépendances dans les deux modèles : (i) dans le modèle de Minkowski, pour un événement donné, seuls les événements situés dans son *cône de lumière futur* peuvent dépendre de lui (*cf.* FIG. 7.1(a)) ; (ii) dans le modèle polyédrique, la résolution des systèmes de contraintes bâtis sur l'équation (4.13) nous donne les transformations respectant la causalité (*cf.* FIG. 7.1(b)).

Nous sommes donc convaincu qu'une formalisation de ce principe, par la définition d'un modèle de temps adéquat, offrirait de nouvelles perspectives pour réunir les modèles polyédrique et synchrone au sein d'un même espace-temps.

[1] Mattern concluait que cette analogie entre le modèle de Minkowski et son propre modèle ne s'étendait pas en dimension trois et plus. La raison en est que Mattern travaille sur un treillis, partiellement ordonné. Il est cependant facile de montrer que son analogie se généralise à n dimensions pour un ensemble *totalement* ordonné.



(a) Principe du cône de lumière dans le modèle de Minkowski.



(b) Rappel de l'exemple d'ordonnancement de la FIG. 4.5.

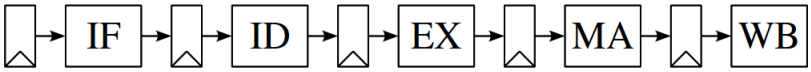
FIG. 7.1: Comparaison entre la vision de l'espace-temps du modèle de Minkowski et un ordonnancement dans le modèle polyédrique.

7.4 Pipeline à étages variables

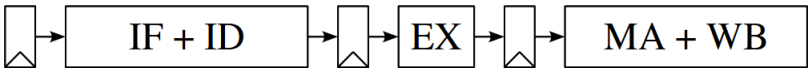
Parmi les techniques basse puissance pour les systèmes sur puce, l'une d'elle consiste à diminuer la fréquence de l'horloge maîtresse lorsque la charge de travail est faible, ou que les performances optimales ne sont pas exigées. On parle alors d'échelonnage dynamique de la fréquence (*dynamic frequency scaling* – DFS). Ainsi, certains signaux peuvent se propager combinatoirement sur des chemins plus longs. Le pipeline à étages variables (*variable stage pipeline*[2] – VSP), consiste à utiliser des court-circuits de registres pour changer dynamiquement les étages d'un pipeline [206]. Un système fonctionnant à plein régime est divisé en de nombreux étages ; les distances parcourues par les signaux sont relativement courtes (pipeline long, *cf.* FIG. 7.2(a)). Lorsque la fréquence d'horloge chute, certains signaux peuvent se propager combinatoirement sur des chemins plus longs ; des étages de pipeline sont alors fusionnés (pipeline court, *cf.* FIG. 7.2(b)).

Plusieurs auteurs présentent le principe du VSP et son intérêt, aussi bien en termes de minimisation du courant de fuite qu'en amélioration du débit [206, 213, 235]. D'autres travaux ont proposé des

[2]Également connu sous le nom de *pipeline stage unification* [213].



(a) Configuration à cinq étages, fréquence d'horloge élevée.



(b) Configuration à trois étages, fréquence d'horloge basse.

FIG. 7.2: Ajustement du nombre et de la longueur des étages dans un pipeline, selon l'exemple de Sasaki *et al.* [206].

algorithmes pour calculer la profondeur de pipeline adéquate pour une charge de travail donnée [76, 236]. Pourtant, à notre connaissance, aucun mécanisme n'avait été proposé pour gérer le passage d'un pipeline court (basse fréquence) à un pipeline long (haute fréquence), et inversement.

Dans Boucaron-Coadou [34], nous avons alors proposé deux implantations du contrôle : l'une basée sur un protocole dynamique, propageant un signal d'arrêt à travers le pipeline ; l'autre utilisant un automate centralisé, contrôlant directement les différents étages. Dans les deux cas, le principe consiste : (i) dans un sens (long vers court), à vider les registres de leurs données avant de les court-circuiter (*cf.* FIG. 7.3(a)) ; (ii) dans l'autre (court vers long), à éliminer les «bulles» du pipeline, c'est-à-dire les données invalides (*cf.* FIG. 7.3(b)).

Ces implantations peuvent encore être améliorées en utilisant une implantation insensible aux latences. En effet, les contrôles distribués et centralisés présentent de nombreux points communs avec, respectivement, les implantations dynamiques et statiques du LID. Ils ne demandent qu'une légère adaptation du protocole.

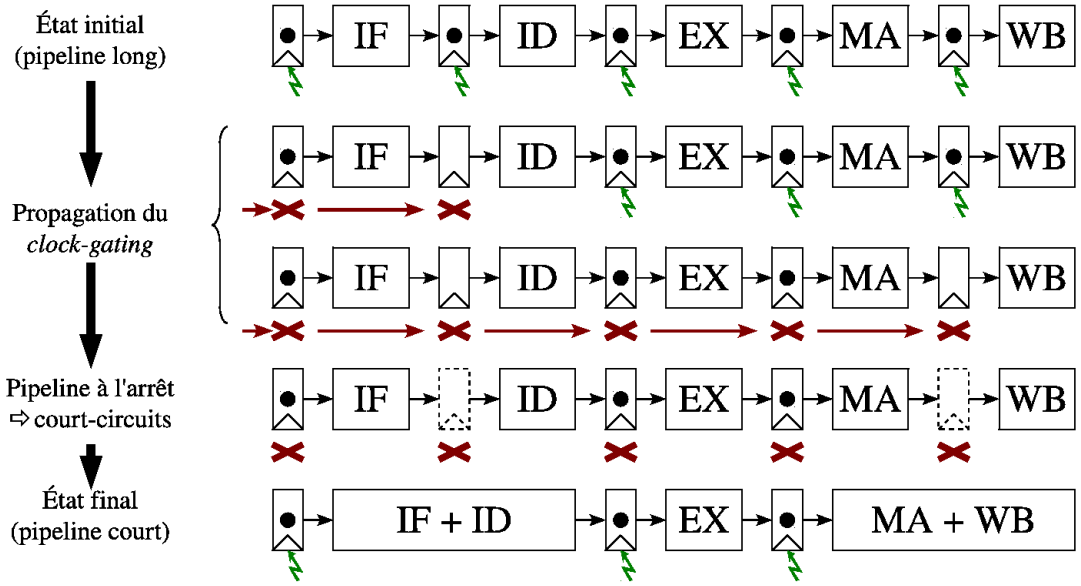
Toutefois, l'*automatisation* de la conception de pipeline à longueur variable demande des efforts de modélisation supplémentaires. *A priori*, le choix des étages pouvant être fusionnés par la chute de la fréquence d'horloge n'est pas quelconque ; cela demande à introduire dans le modèle des informations supplémentaires, de façon à garantir la correction de la transformation. Une idée consisterait à construire les horloges du pipeline court par synchronisation (ou abstraction) de certaines horloges du pipeline long.

7.5 Validation expérimentale

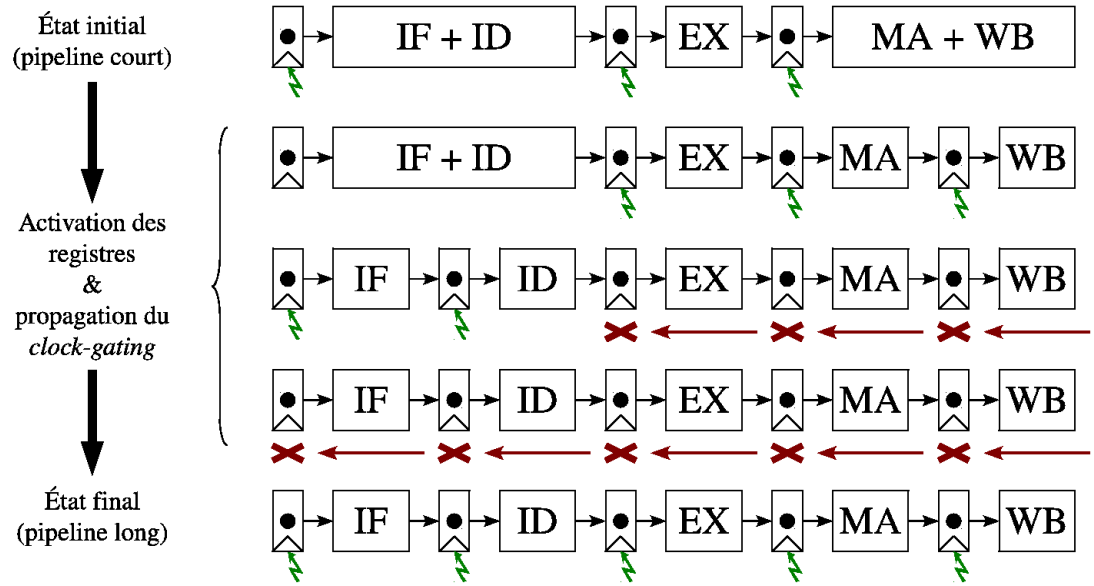
Les transformations sur le modèle et la génération de code pour la conception insensible aux latences sont en cours d'implantation dans la bibliothèque `kpassaSimu`, noyau du logiciel K-Passa [35].

K-Passa était initialement un outil de simulation, dédié à l'analyse et à l'ordonnancement statique de différents réseaux de processus (*e.g.* graphes marqués, graphes SDF, KRG). Dans sa dernière version stable (v.2) implantée par Julien Boucaron, K-Passa est capable de détecter les verrous mortels, résoudre les équations de balances, ou encore ordonnancer le modèle et évaluer ses performances.

À terme, nous comptons y intégrer notre flot de conception, en nous appuyant sur des outils tiers tels que Clan [20, 19] pour la construction du modèle polyédrique, Polylib [120] pour sa manipulation, et Piplib [84] pour résoudre les systèmes de contraintes paramétrés.



(a) Transition de pipeline long à pipeline court : on vide les registres avant de les court-circuiter.



(b) Transition de pipeline court à pipeline long : on remplit les registres pour éliminer les «bulles».

FIG. 7.3: Détail des transitions entre deux fréquences d'horloge.

Annexe A

Notations et rappels mathématiques

> Avec l'ordinateur et les langages de programmation, les mathématiques ont récemment acquis des outils, et leurs notations devraient être réexaminées à cette lumière. L'ordinateur peut, en effet, être utilisé comme un «locuteur natif» patient, précis et maîtrisant la notation mathématique.
>
> Kenneth E. Iverson, *Math for the Layman*, 1999.

Sommaire

A.1 Théorie *n*-synchrone

Dans cette section, nous rappelons les définitions et propriétés des séquences binaires ultimement périodiques et de certains de leurs opérateurs. Ces séquences binaires tirent leur origines des travaux sur la théorie *n*-synchrone (*cf.* sous-section 2.1.3).

A.1.1 Mots et séquences binaires

Définition A.1 (Mot binaire)**.** *Soit* $\mathbb{B} = \{0,1\}$ *l'ensemble des valeurs booléennes, notre alphabet. On note* $\mathbb{B}^*$ *l'ensemble des* mots binaires*, fermeture de Kleene sur* $\mathbb{B}$*.*

$\mathbb{B}^*$ est un monoïde libre, avec le mot vide ε comme élément neutre de la concaténation. Pour tout mot binaire w, on note $|w|$ sa longueur, ainsi que $|w|_0$ et $|w|_1$ le nombre d'occurrences de «0» et «1» respectivement. Soit w_i la $i^{ème}$ lettre de w, avec $i \in \mathbb{N}^*$, pour $w \neq \varepsilon$. On note également $w_{tête} = w_1$ et w_{queue} les mots tels que $w = w_{tête}.w_{queue}$. Par extension de l'algèbre booléenne, les opérateurs de conjonction, disjonction et négation s'appliquent lettre-à-lettre sur des mots de même longueur. La position de la $i^{ème}$ occurrence du booléen b dans le mot binaire w est notée $\text{idx}_b(w,i)$.

Exemple A.2. $\text{idx}_1(0\mathbf{1}01101110,1) = 2$ et $\text{idx}_0(0101101110\mathbf{0},4) = 10$.

Par convention, nous posons $\text{idx}_b(w,i) = +\infty$ si $|w|_b < i$. Enfin, le préfixe de longueur n d'un mot binaire w est noté $\text{prf}(w,n)$, tel que $\text{prf}(w,n) = w_1 \dots w_n$, et on note $\text{suf}(w,n) = w_{n+1} \dots w_{|w|}$ le mot formé par w privé de ses n premières lettres.

Définition A.3 (Débit d'un mot binaire)**.** *Le* débit *d'un mot binaire est défini tel que :* $\forall w \in \mathbb{B}^*$,

$$\text{débit}(w) = \begin{cases} 1 & \text{si } w = \varepsilon \\ \dfrac{|w|_1}{|w|} & \text{sinon} \end{cases} \tag{A.1}$$

Cette définition prend tout son sens si l'on interprète «1» comme étant un instant d'activité, et «0» comme une pause.

Exemple A.4. $\text{débit}(1001101110) = \frac{6}{10}$.

Définition A.5 (Sous-flot)**.** *Soit deux mots binaires u et v de longueur n. On dit que u est un* sous-flot *de v, noté $u \sqsubseteq v$, si et seulement si :*

$$\forall i \in [\![1,n]\!],\ u_i \Rightarrow v_i \tag{A.2}$$

Ce qui peut aussi être exprimé sous la forme :

$$u \sqsubseteq v \ \Leftrightarrow\ \exists w \in \mathbb{B}^n,\ u = v \wedge w \tag{A.3}$$

Proposition A.6. $\forall n \in \mathbb{N},\ \forall u,v \in \mathbb{B}^n$,

$$u \sqsubseteq v \quad \Leftrightarrow \quad \begin{cases} u \wedge v = u \\ u \vee v = v \end{cases} \tag{A.4}$$

$$u \sqsubseteq v \quad \Leftrightarrow \quad \overline{v} \sqsubseteq \overline{u} \tag{A.5}$$

$$(u \sqsubseteq v) \wedge (v \sqsubseteq u) \quad \Leftrightarrow \quad u = v \tag{A.6}$$

$$(u \sqsubseteq v) \wedge (|u|_1 = |v|_1) \quad \Leftrightarrow \quad u = v \tag{A.7}$$

Définition A.7 (Séquence binaire infinie)**.** *Soit $\mathbb{B}^\omega$ l'ensemble des séquences binaires infinies. Une séquence $w \in \mathbb{B}^\omega$ est dite* ultimement périodique *si et seulement si elle est de la forme $u.v^\omega$, avec $u,v \in \mathbb{B}^*$; on nomme u la partie* initiale *de w, et v sa partie* périodique[1]*. On étend la notion de débit aux séquences par* $\text{débit}(w) = \text{débit}(v)$.

On note $\mathbb{P}_p^k$ l'ensemble des séquences binaires ultimement périodiques, ou séquences k-périodiques, *avec une partie périodique de longueur p contenant k occurences de «1». On appelle k la* périodicité *et p la* période *de telles séquences. Enfin, soit $\mathbb{P}$ l'ensemble des séquences ultimement périodiques.*

[1] Nous représentons toujours les séquences ultimement périodiques sous leur forme la plus factorisée. Par exemple, si l'on considère la séquence $011.(0101)^\omega$, nous dirons que sa partie initiale est 01, et sa partie périodique est 10.

Exemple A.8. $01\,(01011)^{\omega} = 0101011...101011010...$ est une séquence binaire 3-périodique de période 5, appartenant à $\mathbb{P}_5^3$. Son débit est $3/5$.

Étant donné qu'il existe une infinité de représentations d'une même séquence, nous ne représentons une séquence (ultimement) périodique que sous sa forme la plus factorisée. Par ailleurs, on peut étendre les notations de préfixe, suffixe et sous-flot aux séquences binaires.

Définition A.9 (Précédence)**.** *Soit u et v deux séquences binaires infinies. On définit une relation de* précédence *établissant un ordre partiel sur $\mathbb{B}^{\omega}$, que l'on note $u \preceq v$ (on dit que u précède v), telle que :*

$$u \preceq v \quad \Leftrightarrow \quad \forall i \in \mathbb{N}^*,\ \mathrm{idx}_1(u,i) \leq \mathrm{idx}_1(v,i) \tag{A.8}$$

Remarque A.10. Les relations de sous-flot et de précédence correspondent à des notions bien distinctes. Un sous-flot traduit une *hiérarchie* ou un échantillonnage de séquence binaire, tandis que la relation de précédence doit être interprétée comme une *antériorité*. Notons toutefois que $u \sqsubseteq v \Rightarrow v \preceq u$.

Définition A.11 (Synchronisabilité)**.** *Soit u et v deux séquences binaires infinies. u et v sont* synchronisables*, que l'on note $u \bowtie v$, si et seulement si :*

$$u \bowtie v \quad \Leftrightarrow \quad \exists i,j \in \mathbb{N},\ u \preceq 0^i.v \wedge v \preceq 0^j.u \tag{A.9}$$

La propriété de synchronisabilité peut également s'exprimer plus simplement par le fait que la différence de position des $i^{ème}$ «1» de u et v est bornée. D'où la proposition suivante.

Proposition A.12. $\forall u,v \in \mathbb{B}^{\omega}$,

$$u \bowtie v \quad \Leftrightarrow \quad \exists d_1,d_2 \in \mathbb{N},\ \forall i \in \mathbb{N}^*,\ -d_1 \leq [u]_i - [v]_i \leq d_2 \tag{A.10}$$

Proposition A.13. $\forall u,v \in \mathbb{P}$,

$$u \bowtie v \quad \Leftrightarrow \quad \text{débit}(u) = \text{débit}(v) \tag{A.11}$$

Cette propriété d'égalité des débits se généralise aux séquences binaires non-périodiques. La notion de débit n'étant pas définie pour de telles séquences, nous étudions le ratio du nombre de «1» des préfixes sur leurs longueurs.

Proposition A.14. $\forall u,v \in \mathbb{B}^{\omega}$,

$$u \bowtie v \quad \Leftrightarrow \quad \lim_{i \to +\infty} \frac{|\mathrm{prf}(u,i)|_1 - |\mathrm{prf}(v,i)|_1}{i} = 0 \tag{A.12}$$

Mandel *et al.* donnent une variante de la définition de la synchronisabilité [157], que nous pouvons traduire avec nos notations par :

$$u \bowtie v \quad \Leftrightarrow \quad \exists b_1,b_2 \in \mathbb{Z},\ \forall i \in \mathbb{N}^*,\ b_1 \leq |\mathrm{prf}(u,i)|_1 - |\mathrm{prf}(v,i)|_1 \leq b_2 \tag{A.13}$$

Ces définitions sont équivalentes.

Proposition A.15. $\forall u,v \in \mathbb{B}^{\omega}$,

$$u \bowtie v \quad \Leftrightarrow \quad \exists b_1,b_2 \in \mathbb{Z},\ \forall i \in \mathbb{N}^*,\ b_1 \leq |\mathrm{prf}(u,i)|_1 - |\mathrm{prf}(v,i)|_1 \leq b_2 \tag{A.14}$$

$$\Leftrightarrow \quad \exists d_1,d_2 \in \mathbb{N},\ \forall i \in \mathbb{N}^*,\ -d_1 \leq \mathrm{idx}_1(u,i) - \mathrm{idx}_1(v,i) \leq d_2 \tag{A.15}$$

A.1.2 Opérateurs *on* et *when*

Dans les chapitres suivants, nous verrons que les ordonnancements des calculs et les routages des données peuvent être modélisés par des séquences binaires. Les transformations opérées sur ces ordonnancements et routages font appel à du filtrage : l'opérateur *on*, défini ci-après, permet d'échantillonner une séquence binaire par une autre pour calculer un sous-flot. Dans ce cadre précis, l'opérateur *when* se comporte comme la réciproque : connaissant la séquence d'origine et son sous-flot, nous pouvons récupérer le filtre.

Par la suite, nous définissons ces deux opérateurs, et prouvons un ensemble de propriétés et d'identités remarquables nécessaires aux preuves des transformations.

Définitions

Définition A.16 (Opérateur *on*)**.** *L'opérateur* on*, noté* $\blacktriangledown$*, est défini récursivement sur les mots binaires, tel que :* $\forall u \in \mathbb{B}^*, \forall v \in \mathbb{B}^{|u|_1}$,

$$\varepsilon \blacktriangledown \varepsilon = \varepsilon \tag{A.16}$$

$$u \blacktriangledown v = \begin{cases} 0.(u_{queue} \blacktriangledown v) & si\ u_{tête} = 0 \\ v_{tête}.(u_{queue} \blacktriangledown v_{queue}) & si\ u_{tête} = 1 \end{cases} \tag{A.17}$$

Ce qui peut être exprimé plus simplement par : $\forall u \in \mathbb{B}^*,\ \forall v \in \mathbb{B}^{|u|_1},\ \exists w \in \mathbb{B}^{|u|}$,

$$u \blacktriangledown v = w = \begin{cases} w_{\mathrm{idx}_1(u,i)} = v_i & \forall i \in [\![1, |v|]\!] \\ w_i = 0 & sinon \end{cases} \tag{A.18}$$

Exemple A.17. $001011 \blacktriangledown 101 = 001001$.

Définition A.18 (Opérateur *when*)**.** *L'opérateur* when*, noté* $\triangle$*, est récursivement défini sur les mots binaires, tel que :* $\forall n \in \mathbb{N},\ \forall u, v \in \mathbb{B}^n$,
si $n = 0$,

$$u \triangle v = \varepsilon \triangle \varepsilon = \varepsilon \tag{A.19}$$

sinon,

$$u \triangle v = \begin{cases} u_{queue} \triangle v_{queue} & si\ v_{tête} = 0 \\ u_{tête}.(u_{queue} \triangle v_{queue}) & si\ v_{tête} = 1 \end{cases} \tag{A.20}$$

Soit, de façon équivalente : $\forall n \in \mathbb{N},\ \forall u, v \in \mathbb{B}^n,\ \exists w \in \mathbb{B}^{|v|_1}$,

$$u \triangle v = w \ \Leftrightarrow\ \forall i \in [\![1, |w|]\!],\ w_i = u_{\mathrm{idx}_1(v,i)} \tag{A.21}$$

Exemple A.19. $001001 \triangle 001011 = 101$.

Intuitivement, la sémantique de ces opérateurs peut être comprise ainsi :

- Dans $u \blacktriangledown v$, nous appliquons les lettres du second opérande v sur les lettres «1» du premier opérande u. D'après la définition récursive (équations A.16 et A.17), la longueur de $u \blacktriangledown v$ croît au fur et à mesure que des lettres de u sont consommées, tandis que v est seulement consommé au «rythme» de $|u|_1$. L'assertion $|v| = |u|_1$ rend la définition consistante.
- À l'inverse, $u \triangle v$ ne préserve que les lettres de u correspondant à des lettres «1» dans v. Ainsi, le résultat est produit à un «rythme» de $|v|_1$, tandis que u et v sont consommées de concert, ce qui revient à échantillonner u par v. Cet opérateur est principalement destiné à être appliqué sous la condition que u soit un sous-flot de v, de façon à ne pas perdre d'information.

Les opérateurs *on* et *when* sont définis sur des mots binaires. Ils peuvent être étendus aux séquences périodiques et ultimement périodiques en restreignant l'étude à leur période. Si les longueurs des préfixes ou des périodes ne correspondent pas, il est toujours possible de «dérouler» les séquences sur une hyperpériode. On impose alors que les séquences soient bien formées, c'est-à-dire qu'il n'y ait pas une différence asymptotique croissante entre les longueurs des mots et le nombre de «1» correspondants, lorsque leurs longueurs tendent vers l'infini. Le résultat est donc une séquence (ultimement) périodique, dont la période est égale au résultat de l'opération appliquée sur les périodes des deux opérandes.

Propriétés de l'opérateur on

Proposition A.20. $\forall u \in \mathbb{B}^*,\ \forall v \in \mathbb{B}^{|u|_1},$

$$|u\blacktriangledown v| = |u| \tag{A.22}$$
$$|u\blacktriangledown v|_1 = |v|_1 \tag{A.23}$$
$$|u\blacktriangledown v|_0 = |u|_0 + |v|_0 \tag{A.24}$$

Proposition A.21 (Débit)**.** $\forall u \in \mathbb{B}^*,\ \forall v \in \mathbb{B}^{|u|_1},$

$$\text{débit}\,(u\blacktriangledown v) = \text{débit}\,(u) \times \text{débit}\,(v) \tag{A.25}$$

Proposition A.22 (Sous-flot)**.** $\forall u \in \mathbb{B}^*,\ \forall v \in \mathbb{B}^{|u|_1},$

$$u\blacktriangledown v \sqsubseteq u \tag{A.26}$$

Corollaire A.23. $\forall u \in \mathbb{B}^*,\ \forall v \in \mathbb{B}^{|u|_1},\ \forall i \in \mathbb{N}^*,$

$$\text{idx}_1\,(u, \text{idx}_1\,(v,i)) = \text{idx}_1\,(u\blacktriangledown v, i) \tag{A.27}$$

Proposition A.24 (Éléments neutres)**.** $\forall u \in \mathbb{B}^*,$
Élement neutre à gauche :

$$1^{|u|}\blacktriangledown u = u \tag{A.28}$$

Élement neutre à droite :

$$u\blacktriangledown 1^{|u|_1} = u \tag{A.29}$$

Proposition A.25 (Éléments idempotents)**.** $\forall n \in \mathbb{N},$

$$1^n\blacktriangledown 1^n = 1^n \tag{A.30}$$

Proposition A.26 (Associativité)**.** $\forall u \in \mathbb{B}^*,\ \forall v \in \mathbb{B}^{|u|_1},\ \forall w \in \mathbb{B}^{|v|_1},$

$$(u\blacktriangledown v)\blacktriangledown w = u\blacktriangledown (v\blacktriangledown w) \tag{A.31}$$

Proposition A.27 (Distributivité à gauche sur opérateurs logiques)**.** $\forall u \in \mathbb{B}^*,\ \forall v,w \in \mathbb{B}^{|u|_1},$

$$u\blacktriangledown (v \wedge w) = (u\blacktriangledown v) \wedge (u\blacktriangledown w) \tag{A.32}$$
$$u\blacktriangledown (v \vee w) = (u\blacktriangledown v) \vee (u\blacktriangledown w) \tag{A.33}$$
$$u\blacktriangledown (v \oplus w) = (u\blacktriangledown v) \oplus (u\blacktriangledown w) \tag{A.34}$$
$$\tag{A.35}$$

Proposition A.28 (Sous-flot (bis))**.** $\forall w \in \mathbb{B}^*,\ \forall u,v \in \mathbb{B}^{|w|_1},$

$$u \sqsubseteq v \ \Leftrightarrow\ w\blacktriangledown u \sqsubseteq w\blacktriangledown v \tag{A.36}$$

Proposition A.29 (Négation)**.** $\forall u \in \mathbb{B}^*,\ \forall v \in \mathbb{B}^{|u|_1},$

$$\overline{u\blacktriangledown v} = \overline{u} \oplus (u\blacktriangledown \overline{v}) \tag{A.37}$$
$$\overline{u\blacktriangledown v} = \overline{u} \vee (u\blacktriangledown \overline{v}) \tag{A.38}$$

Corollaire A.30. $\forall u \in \mathbb{B}^*,\ \forall v \in \mathbb{B}^{|u|_1},$

$$u\blacktriangledown \overline{v} = \overline{u\blacktriangledown v} \wedge u \tag{A.39}$$

Propriétés de l'opérateur when

Proposition A.31 (Élements neutres à droite)**.** $\forall u \in \mathbb{B}^*,$

$$u \triangle 1^{|u|} = u \tag{A.40}$$

Proposition A.32 (Éléments idempotents)**.** $\forall n \in \mathbb{N},$

$$1^n \triangle 1^n = 1^n \tag{A.41}$$

Proposition A.33. $\forall n \in \mathbb{N},\ \forall u,v \in \mathbb{B}^n,$

$$u \sqsubseteq v \ \Rightarrow\ v \triangle u = 1^{|u|_1} \tag{A.42}$$

Proposition A.34 (Débit)**.** $\forall n \in \mathbb{N}^*,\ \forall u,v \in \mathbb{B}^n,\ |v|_1 > 0,$

$$u \sqsubseteq v \ \Rightarrow\ \text{débit}\,(u \triangle v) = \frac{\text{débit}\,(u)}{\text{débit}\,(v)} \tag{A.43}$$

Proposition A.35 (Distributivité à droite sur opérateurs logiques)**.** $\forall n \in \mathbb{N},\ \forall u,v,w \in \mathbb{B}^n,$

$$(u \wedge v) \triangle w = (u \triangle w) \wedge (v \triangle w) \tag{A.44}$$
$$(u \vee v) \triangle w = (u \triangle w) \vee (v \triangle w) \tag{A.45}$$
$$(u \oplus v) \triangle w = (u \triangle w) \oplus (v \triangle w) \tag{A.46}$$
$$\tag{A.47}$$

Proposition A.36 (Sous-flot)**.** $\forall n \in \mathbb{N},\ \forall u,v,w \in \mathbb{B}^n,$

$$u \sqsubseteq v \ \Rightarrow\ u \triangle w \sqsubseteq v \triangle w$$

Proposition A.37 (Négation)**.** $\forall n \in \mathbb{N}^*,\ \forall u,v \in \mathbb{B}^n,$

$$\overline{u \triangle v} = \overline{u} \triangle v \tag{A.48}$$

Proposition A.38. $\forall n \in \mathbb{N}^*,\ \forall u,v \in \mathbb{B}^n,$

$$u \wedge v = 0^{|u|} \ \Rightarrow\ u \triangle (u \vee v) = \overline{v} \triangle (u \vee v) \tag{A.49}$$

Proposition A.39. $\forall n \in \mathbb{N}^*,\ \forall u,v,w \in \mathbb{B}^n,$

$$v \sqsubseteq w \ \Rightarrow\ u \triangle v = (u \wedge w) \triangle v \tag{A.50}$$

Propriétés communes

Les propriétés algébriques des opérateurs *on* et *when* ne permettent pas de les composer aisément. Cet inconvévient demande à prouver un ensemble d'identités remarquables, sur lesquelles reposent les transformations et preuves du chapitre 3.

Proposition A.40. $\forall v \in \mathbb{B}^*,\ \forall u \in \mathbb{B}^{|v|_1},$

$$u = (v \blacktriangledown u) \triangle v \tag{A.51}$$

Proposition A.41. $\forall n \in \mathbb{N}^*,\ \forall u, v \in \mathbb{B}^n,$

$$u \sqsubseteq v \ \Leftrightarrow\ u = v \blacktriangledown (u \triangle v) \tag{A.52}$$

Proposition A.42. $\forall n \in \mathbb{N}^*,\ \forall u, v \in \mathbb{B}^n,\ \forall w \in \mathbb{B}^{|u \wedge v|_1},$

$$(u \triangle v) \blacktriangledown w = ((u \wedge v) \blacktriangledown w) \triangle v \tag{A.53}$$

Proposition A.43. $\forall u \in \mathbb{B}^*,\ \forall v, w \in \mathbb{B}^{|u|_1},$

$$(u \blacktriangledown v) \triangle (u \blacktriangledown w) = v \triangle w \tag{A.54}$$

Proposition A.44. $\forall n \in \mathbb{N}^*,\ \forall u, v, w \in \mathbb{B}^n,$

$$u \triangle (v \wedge w) = (u \triangle w) \triangle (v \triangle w) \tag{A.55}$$

Proposition A.45. $\forall v, w \in \mathbb{B}^*,\ \forall u \in \mathbb{B}^{|w|_1},$

$$u \triangle (v \triangle w) = (w \blacktriangledown u) \triangle (v \wedge w) \tag{A.56}$$

Proposition A.46. $\forall n \in \mathbb{N},\ \forall u, w \in \mathbb{B}^n,\ \forall v \in \mathbb{B}^{|u|_1},$

$$(u \blacktriangledown v) \wedge w = (u \wedge w) \blacktriangledown (v \triangle (w \triangle u)) \tag{A.57}$$

Proposition A.47. $\forall n \in \mathbb{N},\ \forall u, w \in \mathbb{B}^n,\ \forall v \in \mathbb{B}^{|u|_1},$

$$(u \blacktriangledown v) \triangle w = (u \triangle w) \blacktriangledown (v \triangle (w \triangle u)) \tag{A.58}$$

Proposition A.48. $\forall n \in \mathbb{N},\ \forall u, v \in \mathbb{B}^n,\ \forall w \in \mathbb{B}^{|v|_1},$

$$u \triangle (v \blacktriangledown w) = (u \triangle v) \triangle w \tag{A.59}$$

Proposition A.49. $\forall n \in \mathbb{N},\ \forall u, v, w \in \mathbb{B}^n,$

$$u \triangle w = v \triangle w\ \Rightarrow\ u \wedge w = v \wedge w \tag{A.60}$$

A.2 Théorie des polyèdres

A.2.1 Définitions

Définition A.50 (Polyèdre, polytope)**.** *Un polyèdre rationnel (convexe) P est un sous-espace de $\mathbb{Q}^d$, défini par l'une des deux représentations (duales) suivantes :*

– *Soit par un système d'inégalités affines, tel que pour $A \in \mathbb{Q}^{m\times d}$ et $\vec{b} \in \mathbb{Q}^m$:*

$$P = \left\{ \vec{x} \in \mathbb{Q}^d \;\middle|\; A\cdot\vec{x} + \vec{b} \geq \vec{0} \right\} \tag{A.61}$$

– *Soit comme l'enveloppe convexe d'un système générateur, formé de trois matrices rationnelles L, R et S telles que :*

$$P = \left\{ \vec{x} \in \mathbb{Q}^d \;\middle|\; \vec{x} = L\cdot\vec{\lambda} + R\cdot\vec{\mu} + S\cdot\vec{\nu},\;\; \lambda_i \in \mathbb{Q},\;\; \mu_i, \nu_i \in \mathbb{Q}^+,\;\; \sum_i \nu_i = 1 \right\} \tag{A.62}$$

Les familles des vecteurs colonnes $\left(\overrightarrow{L_1}\dots\overrightarrow{L_j}\right)$, $\left(\overrightarrow{R_1}\dots\overrightarrow{R_k}\right)$ et $\left(\overrightarrow{S_1}\dots\overrightarrow{S_l}\right)$ sont respectivement les ensembles de lignes, rayons et sommets de P.

Un polytope est un polyèdre borné [169].

La dualité des deux représentations fut prouvée par Motzkin *et al.* [172]. Le passage d'une forme à l'autre est possible par l'algorithme de Chernikova [59, 142]. Pour cela, les polyèdres sont représentés sous leur forme *homogène* [169, 232]. Par extension, nous pouvons considérer le cas concave par décomposition en une union de polyèdres convexes distincts [234].

La représentation homogène d'un polyèdre est plus adaptée aux outils d'algèbre linéaire et à l'automatisation des calculs. La représentation homogène du polyèdre (A.61) est :

$$P = \left\{ \begin{pmatrix}\vec{x}\\1\end{pmatrix} \in \mathbb{Q}^{d+1} \;\middle|\; \begin{pmatrix}A & -\vec{b}\\ \vec{0} & 1\end{pmatrix}\cdot\begin{pmatrix}\vec{x}\\1\end{pmatrix} \geq \vec{0} \right\} \tag{A.63}$$

Le même principe s'applique à la forme génératrice du polyèdre (A.62) :

$$P = \left\{ \begin{pmatrix}\vec{x}\\1\end{pmatrix} \in \mathbb{Q}^{d+1} \;\middle|\; \begin{pmatrix}\vec{x}\\1\end{pmatrix} = \begin{pmatrix}L\\ \vec{0}\end{pmatrix}\cdot\vec{\lambda}' + \begin{pmatrix}R & V\\ \vec{0} & \vec{1}\end{pmatrix}\cdot\vec{\mu}',\;\; \mu'_i \in \mathbb{Q}^+ \right\} \tag{A.64}$$

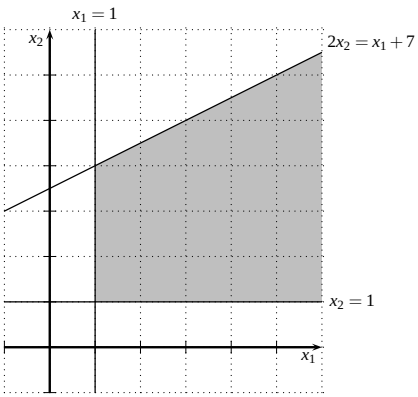
Exemple A.51. Considérons le polyèdre de la FIG. A.1(a), défini par le système de contraintes suivant :

$$P = \left\{ \vec{x} \in \mathbb{Q}^2 \;\middle|\; \begin{pmatrix}1 & 0\\ 0 & 1\\ 1 & -2\end{pmatrix}\cdot\begin{pmatrix}x_1\\x_2\end{pmatrix} + \begin{pmatrix}-1\\-1\\7\end{pmatrix} \geq \vec{0} \right\} \tag{A.65}$$

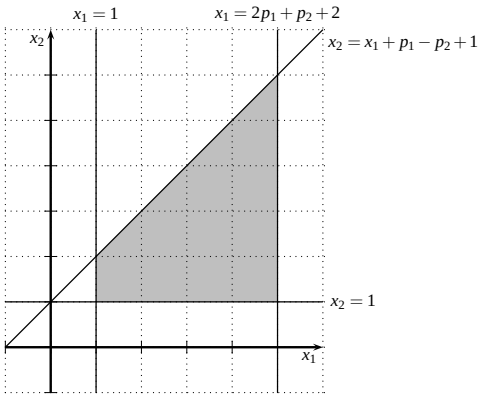
Sa représentation sous forme de système générateur est la suivante :

$$P = \left\{ \vec{x} \in \mathbb{Q}^2 \;\middle|\; \vec{x} = \begin{pmatrix}1 & 2\\ 0 & 1\end{pmatrix}\cdot\vec{\mu} + \begin{pmatrix}1 & 1\\ 1 & 4\end{pmatrix}\cdot\vec{\nu},\;\; \mu_i, \nu_i \in \mathbb{Q}^+,\;\; \sum_i \nu_i = 1 \right\} \tag{A.66}$$

Les polyèdres paramétrés dépendent, par définition, d'un ensemble de paramètres. Dans ce manuscrit, nous ne considérons que des polyèdres *linéairement* paramétrés, où seule la partie constante dépend de paramètres. Par la suite, nous les appelons simplement polyèdres paramétrés.



(a) Polyèdre non-borné à deux dimensions.



(b) Polytope paramétré à 2 dimensions, avec $p_1 = 1$ et $p_2 = 1$.

FIG. A.1: Exemples de polyèdre et polytope.

Définition A.52 (Polyèdre paramétré)**.** *Un polyèdre rationnel (convexe) linéairement paramétré $P_{\vec{p}}$, selon les n paramètres de $\vec{p}$, est défini sous la forme d'un système d'inégalités telles que :*

$$P_{\vec{p}} = \left\{ \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} \in \mathbb{Q}^{d+n} \;\middle|\; \begin{pmatrix} A_X & A_P \end{pmatrix} \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} + \vec{b} \geq \vec{0} \right\} \tag{A.67}$$

où $A_X \in \mathbb{Q}^{m\times d}$, $A_P \in \mathbb{Q}^{m\times n}$, $\vec{b} \in \mathbb{Q}^m$, et $\vec{p} \in \mathbb{Q}^n$. Par le théorème de Loechner-Wilde [154], sa représentation duale est de la forme :

$$P_{\vec{p}} = \left\{ \vec{x} \in \mathbb{Q}^d \;\middle|\; \vec{x} = L\vec{\lambda} + R\vec{\mu} + S(\vec{p})\vec{\nu},\ \ \lambda_i \in \mathbb{Q},\ \ \mu_i, \nu_i \in \mathbb{Q}^+,\ \ \sum_i \nu_i = 1 \right\} \tag{A.68}$$

où L et R sont les matrices rationnelles constantes des lignes et rayons du polyèdre, et $S(\vec{p})$ est la matrice rationnelle de ses sommets, dont chaque coefficient est une fonction affine de $\vec{p}$.

Exemple A.53. Considérons le polytope paramétré $P_{\vec{p}}$, défini par :

$$P_{\vec{p}} = \left\{ \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} \in \mathbb{Q}^{2+2} \;\middle|\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 2 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix} + \begin{pmatrix} -1 \\ 2 \\ -1 \\ 1 \end{pmatrix} \geq \vec{0} \right\} \tag{A.69}$$

La représentation graphique de $P_{\vec{p}}$ est donnée en FIG. A.1(b), pour $\vec{p} = (1,1)$.

Définition A.54 (Réseau entier)**.** *Un réseau[2] entier Λ de $\mathbb{Z}^d$ est un sous-groupe additif discret de $\mathbb{Z}^d$, de la forme :*

$$\Lambda = \left\{ A\vec{x} + \vec{b} \;\middle|\; \vec{x} \in \mathbb{Z}^d \right\} \tag{A.70}$$

où $A \in \mathbb{Z}^{d\times d}$ est une matrice dont les colonnes $\overrightarrow{A_1}, \overrightarrow{A_2}, \ldots \overrightarrow{A_d}$ forment une base de $\mathbb{Z}^d$, et $\vec{b} \in \mathbb{Z}^d$.

[2]Utilisé au sens mathématique du terme, en particulier en géométrie et théorie des groupes. De nombreux auteurs préfèrent l'anglicisme *lattice* pour éviter les confusions.

Définition A.55 ($\mathbb{Z}$-polyèdre, $\mathbb{Z}$-polytope)**.** *Un $\mathbb{Z}$-polyèdre est l'intersection d'un polyèdre rationnel P et d'un réseau entier Λ. Un $\mathbb{Z}$-polyèdre de dimension d est dit standard si $\Lambda = \mathbb{Z}^d$. Un $\mathbb{Z}$-polytope est un $\mathbb{Z}$-polyèdre borné.*

Un $\mathbb{Z}$-polyèdre s'exprime sous la forme d'un réseau linéairement borné (*linearly bounded lattice*), image affine d'un polyèdre [94] :

$$Z = \left\{ A\vec{x} + \vec{b} \;\middle|\; C\vec{x} + \vec{d} \geq \vec{0},\ \vec{x} \in \mathbb{Z}^d \right\} \tag{A.71}$$

où $\Lambda = \left\{ A\vec{x} + \vec{b} \;\middle|\; \vec{x} \in \mathbb{Z}^d \right\}$ est le réseau, et $P = \left\{ \vec{x} \in \mathbb{Z}^d \;\middle|\; C\vec{x} + \vec{d} \geq \vec{0} \right\}$ est le *polyèdre de coordonnées*.

Le modèle polyédrique est très expressif, et supporte un large éventail de transformations algébriques, parmi lesquelles :

- l'*intersection* : $\bigcap_i Z_i = \bigcap_i (P_i \cap \Lambda_i) = (\bigcap_i P_i) \cap (\bigcap_i \Lambda_i)$.
- la *différence* : un $\mathbb{Z}$-polyèdre se décompose en l'union disjointe des $\mathbb{Z}$-polyèdres qu'il inclut. Ainsi, $Z \setminus Z' = (\bigcup_i Z_i) \setminus \left(\bigcup_j Z'_j \right) = \bigcup_i \left(\bigcap_j \left(Z_i \setminus Z'_j \right) \right)$.
- l'*union disjointe* : il est souvent difficile de construire l'union de $\mathbb{Z}$-polyèdres ; leur union disjointe consiste à les séparer en un ensemble de $\mathbb{Z}$-polyèdres deux à deux disjoints par des intersections et différences successives. Dans le cas général, l'union n'est pas convexe.
- la *compression* : cas particulier du changement de base, un $\mathbb{Z}$-polyèdre quelconque est transformé en un $\mathbb{Z}$-polyèdre standard, en supprimant les «trous» entre ses points à pas réguliers.
- le *calcul d'image et de pré-image* par une fonction affine : notons que le calcul d'une pré-image par une fonction affine demande à ce que cette fonction soit inversible.

Les détails des transformations et des calculs sont disponibles dans les travaux de Quinton *et al.* [198], Gautam et Rajopadhye [94] et Seghir [210].

A.2.2 Dénombrement de points à coordonnées entières

Comme nous le verrons par la suite, il est crucial de pouvoir dénombrer les points à coordonnées entières inclus dans un polyèdre. Les travaux d'Ehrhart [80] sont pionniers en la matière.

Définition A.56 (Nombre n-périodique)**.** *Un nombre n-périodique rationnel $U(p)$ est une fonction $\mathbb{Z}^n \to \mathbb{Q}$, telle qu'il existe une multi-période $q = (q_1, ..., q_n) \in \mathbb{N}^n$, telle que $U(p) = U(p')$ quel que soit $p_i \equiv p'_i \mod q$, pour $1 \leq i \leq n$. Le plus petit commun multiple des q_i est la période de $U(p)$.*

Définition A.57 (Quasi-polynôme)**.** *Un quasi-polynôme en $\vec{p} \in \mathbb{Z}^n$ de degré d est une fonction $f : \mathbb{Z}^n \to \mathbb{Q}$, définie par :*

$$f(\vec{p}) = \sum_{i_1=0}^{d} \cdots \sum_{i_n=0}^{d} c_{i_1,...,i_n} p_1^{i_1} \cdots p_n^{i_n} \tag{A.72}$$

où chaque $c_{i_1,...,i_n}$ est un nombre multidimensionnel, au plus n-périodique. Le plus petit commun multiple des périodes des cœfficients $c_{i_1,...,i_n}$ est la période du quasi-polynôme.

Définition A.58 (Dénominateur d'un polyèdre)**.** *Le dénominateur relatif à un paramètre $\vec{p}$ d'un polytope est le plus petit commun multiple (ppcm) des cœfficients de $\vec{p}$ qui apparaissent dans les expressions affines des sommets, pour toute valeur entière des paramètres. Le dénominateur d'un polyèdre est le ppcm de ses dénominateurs relatifs aux paramètres.*

Théorème A.59 (Théorème de Clauss)**.** *Le nombre de points entiers dans un d-polytope paramétré quelconque $P_{\vec{p}}$, avec $\vec{p} \in \mathbb{Z}^n$, peut être donné par un ensemble fini de quasi-polynômes en $\vec{p}$ de degré d. Chaque quasi-polynôme est valide sur un sous-ensemble de valeurs des paramètres, dit* domaine de validité. *La période relative à un paramètre p_i du quasi-polynôme sur un domaine de validité donné est égale au dénominateur relatif à p du polytope. Lorsque le polytope est entier, sur un domaine de validité donné, le nombre de points entiers correspondant est donné par un simple polynôme.*

Exemple A.60. Le nombre de points à coordonnées entières contenus dans le polytope de l'exemple A.53 est donné par le quasi-polynôme suivant :

$$\begin{cases} 4p_1^2 - \dfrac{p_2^2}{2} + p_1p_2 + 9p_1 + \dfrac{3p_2}{2} + 5 & \text{si} \quad 2p_2 + p_2 + 1 \geq 0 \ \wedge \ p_1 - p_2 + 1 \geq 0 \\ \dfrac{9p_1^2}{2} + \dfrac{21p_1}{2} + 6 & \text{si} \quad p_1 + \dfrac{2}{3} \geq 0 \ \wedge \ -p_1 + p_2 - 1 \geq 0 \end{cases} \tag{A.73}$$

Pour plus de détails quant aux algorithmes de calcul des quasi-polynômes d'Ehrhart, nous renvoyons le lecteur aux travaux récents de Seghir [210] et Verdoolaege [229].

A.3 Graphes de permutation et d'intervalle

A.3.1 Graphe de permutation

Définition A.61 (Graphe de permutation)**.** *Soit $\pi = [\pi_1, \pi_2, ..., \pi_n]$ une permutation du n-uplet $[1, 2, ..., n]$. Notons π_i^{-1} la position dans la permutation du $i^{ème}$ élément du n-uplet donné. Le graphe de permutation $G[\pi] = (V, E)$ est un graphe orienté tel que :*

– *Chaque sommet correspond à un élément de la permutation :*

$$V \stackrel{\text{def}}{=} \{1, 2, ..., n\} \tag{A.74}$$

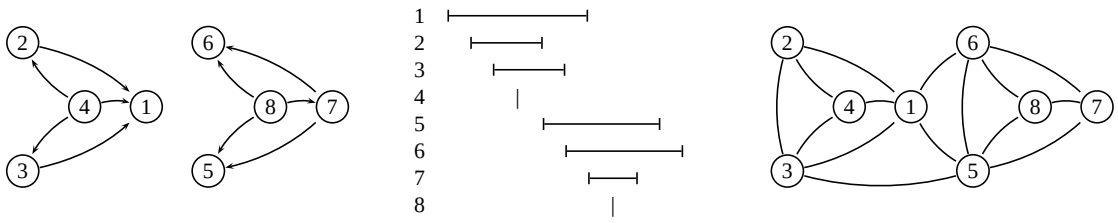
– *Il existe un arc d'un sommet i vers un sommet j si et seulement si leurs jetons correspondants ont été mutuellement permutés :*

$$E \stackrel{\text{def}}{=} \left\{ (i, j) \mid i, j \in V, \ i > j, \ \pi_i^{-1} < \pi_j^{-1} \right\} \tag{A.75}$$

Nous pouvons l'exprimer plus simplement par : le $i^{ème}$ jeton entre dans le mélangeur après le $j^{ème}$, et en sort avant.

Ainsi, le degré entrant $d^-(s)$ d'un sommet s est égal au nombre de fois que le jeton correspondant sera dépassé par d'autres jetons. Le degré sortant $d^+(s)$ est quant à lui égal au nombre de fois que son jeton correspondant dépassera d'autres jetons. Nous remarquons qu'un tel graphe est acyclique ; deux jetons ne peuvent se doubler à tour de rôle.

L'algorithme de coloration de Supowit [101, 222] donne une χ-coloration minimale, en un temps séquentiel de $O(n \log_2 n)$. Des algorithmes de coloration parallèle ont également été développés, dont le plus performant, à notre connaissance, s'exécute en un temps de l'ordre de $O\left(\log_2^2 n\right)$ sur $O\left(n^2 / \log_2 n\right)$ processeurs d'une PRAM CREW [180].



(a) Graphe de permutation.

(b) Interférences de la permutation.

(c) Graphe d'intervalle.

FIG. A.2: Exemple de permutation $\pi = [4,2,3,1,8,7,5,6]$: lien avec le graphe d'intervalle.

A.3.2 Code de Lehmer

En analyse combinatoire, le code de Lehmer [146] est défini comme suit :

Définition A.62 (Code de Lehmer)**.** *Le code de Lehmer associe à une permutation π des éléments de $[\![1,n]\!]$ l'application $L(\pi)$ définie sur $[\![1,n]\!]$ par :*

$$L(\pi,i) = \text{card}\left\{ j \;\middle|\; i < j,\ \pi_i > \pi_j \right\} \tag{A.76}$$

Notons que l'application L est bijective : pour toute permutation π, il existe un et un seul code de Lehmer $L(\pi)$. Un algorithme simple permet de déterminer π en fonction de $L(\pi)$. En effet, π_i est égal au $(L(\pi,i)+1)^{\text{ème}}$ élément de $[1,\ldots,n] \setminus \bigcup_{j=1}^{i-1} \{\pi_j\}$.

Exemple A.63. Considérons le code de Lehmer $L(\pi) = [2,1,2,1,0]$, et déroulons l'algorithme :
- π_1 est le $3^{\text{ème}}$ élément de $[1,2,3,4,5]$, donc $\pi_1 = 3$.
- π_2 est le $2^{\text{ème}}$ élément de $[1,2,4,5]$, donc $\pi_2 = 2$.
- π_3 est le $3^{\text{ème}}$ élément de $[1,4,5]$, donc $\pi_3 = 5$.
- π_4 est le $2^{\text{ème}}$ élément de $[1,4]$, donc $\pi_4 = 4$.
- π_5 est le 1^{er} élément de $[1]$, donc $\pi_5 = 1$.

Nous concluons alors que $\pi = [3,2,5,4,1]$.

A.3.3 Graphe d'intervalle

Définition A.64 (Graphe d'intervalle)**.** *Soit $I_1, I_2, ..., I_n \subset \mathcal{R}$ un ensemble d'intervalles sur l'ensemble ordonné $\mathcal{R}$. Le graphe d'intervalle $g = (V,E)$ correspondant est un graphe non-orienté, ayant un sommet pour chaque intervalle de l'ensemble, et un arc entre chaque paire de sommets dont les intervalles s'intersectent, i.e. :*

$$V = \{I_1, I_2, ..., I_n\} \tag{A.77}$$

$$E = \left\{ I_i \cap I_j \neq \emptyset \;\middle|\; i,j \in [\![1..n]\!],\ i \neq j \right\} \tag{A.78}$$

Exemple A.65. Considérons la permutation $\pi = [4,2,3,1,8,7,5,6]$, dont le graphe de permutation est représenté en FIG. A.2(a). La FIG. A.2(b) correspond aux interférences entre jetons, considérant qu'un nouveau jeton entre dans le mélangeur à chaque pas d'exécution, et que les jetons sortent du mélangeur au plus tôt. Leur représentation sous forme de graphe est donnée en FIG. A.2(c).

Tout graphe d'intervalle $G = (E,V)$ est triangulé [101], et peut donc être coloré en un temps de l'ordre de $O(|E|+|V|)$, permettant ainsi le développement d'un certain nombre d'heuristiques. Le lecteur pourra se référer à [178] pour un état de l'art détaillé.

Annexe B

Preuves

Une théorie a la seule alternative d'être vraie ou fausse. Un modèle a une troisième possibilité : il peut être correct, mais non pertinent.

Manfred Eigen, The Origin of Biological Information. *The Physicist's Conception of Nature*, J. Mehra (éd.), 1973.

Sommaire

B.1 Preuves du chapitre 2

Les preuves des lemmes et théorèmes 2.4 à 2.9 sont données par Commoner *et al.* [66].

Preuve du théorème 2.18. Les nœuds d'un graphe marqué temporisé ne peuvent être exécutés à une fréquence supérieure à un tel débit [201]. Les exécutions peuvent atteindre ce débit [14]. □

Preuve du théorème 2.23. Par la définition 2.17, le débit $\Theta(n)$ d'un nœud n est tel que :

$$\Theta(n) = \lim_{i \to +\infty} \frac{i}{t_n(i)} \tag{B.1}$$

où $t_n(i)$ est le temps écoulé au $i^{\text{ème}}$ tirage de n. Pour tout circuit c d'un graphe connexe g, chaque nœud ne peut être exécuté plus d'une fois par instant ; cela implique que pour tout nœud n de c, $i \leq t_n(i)$. Donc le débit global est d'au plus 1, et nous ignorons les cas où $\sum_{p \in \mathcal{P}'_c} M'_0(p)$ est supérieur à $\sum_{n \in \mathcal{N}'_c} L'(n) + \sum_{p \in \mathcal{P}'_c} L'(p)$.

Nous considérons désormais les cycles c tels que $\sum_{p \in \mathcal{P}'_c} M'_0(p)$ est inférieur ou égal à $\sum_{n \in \mathcal{N}'_c} L'(n) + \sum_{p \in \mathcal{P}'_c} L'(p)$. Un ordonnancement ultimement périodique peut être calculé pour un graphe marqué avec la règle de tirage ASAP. De plus, comme mentionné précédemment, toutes les règles de tirages

sont compatibles entre elles. Ainsi, nous pouvons borner l'étude sur une période. Le temps minimum pour qu'un jeton fasse le tour du cycle c pour revenir dans sa position initiale est égal à la somme des latences, soit $\sum_{n\in\mathcal{N}'_c} L'(n)+\sum_{p\in\mathcal{P}'_c} L'(p)$. Cette somme des latences des nœuds et places est elle-même égale à la longueur d'une période. Sur une période, pour que tous les jetons fassent le tour du circuit, chaque nœud est tiré $\sum_{p\in\mathcal{P}'_c} M'_0(p)$ fois (par lemme 2.4 et théorème 2.7).

Pour résumer, le débit d'un circuit c est :

$$\Theta(c) = \min\left(\frac{\sum\limits_{p\in\mathcal{P}'_c} M'_0(p)}{\sum\limits_{n\in\mathcal{N}'_c} L'(n)+\sum\limits_{p\in\mathcal{P}'_c} L'(p)}, 1\right) \tag{B.2}$$

Soit g' l'équivalent complémenté de g. Puisque g est un graphe connexe (éventuellement acyclique), alors g' est fortement connexe à cause des places complémentaires, et $\Theta(g)=\Theta(g')$. Le débit du graphe fortement connexe est égal au débit de son circuit le plus lent (par théorème 2.18). □

B.2 Preuves du chapitre 3

Preuve de la proposition 3.8. Propriété 8 de Boucaron [33]. □

Preuve de la proposition 3.16.

1. Il ne peut exister d'auto-dépendance dans un GDE, donc le point 1 est faux pour $id_{\mathcal{M}}$. Cependant, le point 2 est vérifié : il est évident que $p_i \xrightarrow{n}_f q_j \Leftrightarrow p_i \xrightarrow{n}_f q_j$ et $\lambda(n)=\lambda(n)$ et $q_j \mathcal{R}_n q_j$ et $p_i, p_i \notin \mathcal{I}$. Ainsi $id_{\mathcal{M}}$ vérifie la définition.
2. Le point 1 est évidemment vrai pour $\mathcal{R}_n^{-1}$: s'il existe un chemin entre p_i et $p'_{i'}$, ce même chemin est entre $p'_{i'}$ et p_i. Aussi, les égalités et équivalences sont symétriques, telles que :

$$p'_{i'}\mathcal{R}_n p_i \;\Rightarrow\; \left(p'_{i'} \xrightarrow{n'}_f q'_{j'} \Leftrightarrow p_i \xrightarrow{n}_f q_j \;\wedge\; \lambda(n')=\lambda(n) \;\wedge\; q'_{j'}\mathcal{R}_n q_j \;\wedge\; p'_{i'}, p_i \notin \mathcal{I}\right)$$

   donc le point 2 est vrai lui aussi, et $\mathcal{R}_n^{-1}$ vérifie la définition.

□

Preuve de la proposition 3.17. Prouvons que la relation $\overset{f}{\sim}$ vérifie les trois propriétés d'une relation d'équivalence, à savoir la réflexivité, la symétrie et la transitivité.

- *Réflexivité* : pour tout $p_i \in \mathcal{M}$, $p_i \overset{f}{\sim} p_i$, par la proposition 3.16 (1).
- *Symétrie* : pour tout $p_i, p_j \in \mathcal{M}$, si $p_i \overset{f}{\sim} p_j$, alors $p_j \overset{f}{\sim} p_i$ par la proposition 3.16 (2).
- *Transitivité* : la relation $\overset{f}{\sim}$ est, par définition, une clôture transitive.

□

Preuve de la proposition 3.18. Par la définition 3.15, nous savons que la moitié de cette proposition est vraie. Nous prouvons maintenant l'implication inverse. Par exemple, nous pouvons définir une relation $\mathcal{R}$ telle que $p_i \mathcal{R} p'_{i'}$ si et seulement si :

$$p_i \xrightarrow{n}_f q_j \Leftrightarrow p'_{i'} \xrightarrow{n'}_f q'_{j'} \;\wedge\; \lambda(n)=\lambda(n') \;\wedge\; q_j \overset{f}{\sim} q'_{j'} \;\wedge\; p_i, p'_{i'} \notin \mathcal{I} \tag{B.3}$$

Ainsi :

$$p_i \overset{f}{\sim} p'_{i'} \;\Rightarrow\; p_i \mathcal{R} p'_{i'} \tag{B.4}$$

Soit $p_i \xrightarrow{n}_f q_j$. Par (B.3), nous savons qu'il existe $q'_{j'}$, tel que $p'_{i'} \xrightarrow{n'}_f q'_{j'}$ et $\lambda(n) = \lambda(n')$ et $q_j \overset{f}{\sim} q'_{j'}$ et $p_i, p'_{i'} \notin \mathcal{I}$. Puis par (B.4), $q_j \mathcal{R} q'_{j'}$. D'où :

$$p_i \overset{f}{\sim} p'_{i'} \;\Rightarrow\; p_i \mathcal{R} p'_{i'} \;\Rightarrow q_j \overset{f}{\sim} q'_{j'} \;\Rightarrow\; q_j \mathcal{R} q'_{j'} \tag{B.5}$$

Nous en concluons que l'équivalence est vraie. □

Preuve du lemme 3.19. Ce lemme est une conséquence directe de l'abstraction d'un KRG en graphe SDF, par la règle 3.7 et le corollaire 3.8. En résolvant les équations de balance nous calculons, pour tout nœud de routage $n \in \mathcal{N}_f \cup \mathcal{N}_s$, le nombre de périodes i_n de n à tirer, de telle façon qu'une période de l'ensemble du graphe soit balancée. De plus, n est tiré $|R(n)|$ fois au cours de chacune de ses périodes. Aussi, posons $j_n = i_n \times |R(n)|$ le nombre de fois que n doit être tiré au cours d'une période balancée du KRG.

Par hypothèse, $R(n)$ est k-périodique de période $|R(n)|$. Il est alors évident que $R(n)$ privé de ses $i_n \times |R(n)|$ premières lettres est égal à lui-même. Donc $R(n) = \text{suf}(R(n), j_n)$. Finalement, l'équation $R(n) = \text{suf}(R(n), kj_n)$ est prouvée par récurrence sur k. □

Preuve de la proposition 3.20. Par la définition 3.1, chaque nœud de routage a une phase initiale, éventuellement vide. Notons l_n la longueur de la phase initiale du nœud de routage n. Si chaque nœud de routage n peut être tiré au moins l_n fois, alors l'ensemble du graphe entre en routage périodique. Sinon, s'il existe un tel nœud ne pouvant être tiré l_n fois, alors tous les chemins passant par n seront ultimement bloqués, et peuvent être ignorés si l'on considère le comportement du graphe au cours d'une séquence de tirages suffisamment longue. Il existe donc $l \geq l_n$, pour tout n, tel que si chaque nœud est exécuté l fois, alors il se trouve en phase périodique.

Considérons le comportement du graphe au cours de cette phase périodique : lorsque la longueur de la séquence de tirage tend vers l'infini, le nombre de jetons transitant par une place reste soit constant (borné), soit tend lui-même vers l'infini. Par le lemme 3.19, nous pouvons déterminer la longueur d'une période du graphe j. Nous pouvons également borner notre étude à la première période du graphe, sachant que son comportement au cours des suivantes sera identique. Selon la proposition 3.18, $p_{l+i} \overset{f}{\sim} p_{l+i+kj}$ si et seulement si :

$$p_{l+i} \rightarrow q_x \Leftrightarrow p_{l+i+kj} \rightarrow q_y \;\text{ et }\; q_x \overset{f}{\sim} q_y \tag{B.6}$$

Ainsi, selon la définition 3.11, il existe deux types de dépendances : les dépendances de flot et les dépendances de sortie. Prouvons (B.6) dans les deux cas.

Dépendance de sortie : $\forall p_i, p_{i+1} \in \mathcal{M},\; p_i \rightarrow_s p_{i+1}$. Soit $q = p$, $x = l+i+1$ et $y = x+kj$. L'équivalence est vraie par récurrence pour tout j et k.

Dépendance de flot : $\forall p, q \in \mathcal{P},\; p\bullet = \bullet q,\; i \curvearrowright_q \circ \curvearrowright_{p\bullet} j \;\Leftrightarrow\; p_i \xrightarrow{p\bullet}_f q_j$. Puis, selon $p\bullet$:

– si $p\bullet \in \mathcal{N}_\lambda \cup \mathcal{N}_c$, alors pour tout i, $l+i \curvearrowright_q \circ \curvearrowright_{p\bullet} l+i+M(q)$, et $p_{l+i} \rightarrow_f q_{l+i+M(q)}$. Soit $x = l+i+M(q)$ et $y = x+kj$; l'équivalence est vraie par récurrence pour tout j et k.

– si $p\bullet \in \mathcal{N}_f$, alors pour tout i :

$$l+i \curvearrowright_q \circ \overset{(b)}{\curvearrowright_{p\bullet}} |\text{prf}(R(p\bullet), l+i)|_b + M(q) \tag{B.7}$$

où $b \in \mathbb{B}$ est l'étiquette associée à l'entrée de $p\bullet$ reliée à p. Soit $i = k'|R(p\bullet)| + i'$ et $i' < |R(p\bullet)|$. Décomposons $R(p\bullet)$ comme suit :

$$|\text{prf}(R(p\bullet), l+i)|_b = |\text{prf}(R(p\bullet), l)|_b + k'|(p\bullet)|_b + |w|_b \tag{B.8}$$

où w est le suffixe de $\text{prf}(R(p\bullet), l+i)$ de longueur i'. Soit :

$$x = |\text{prf}(R(p\bullet), l)|_b + k'|(p\bullet)|_b + |w|_b + M(q) \quad \text{(B.9)}$$

$$y = x + \frac{|R(p\bullet)|_b}{|R(p\bullet)|} kj \quad \text{(B.10)}$$

L'équivalence est vérifiée par récurrence pour tout k, et pour tout j de la forme $k''|R(p\bullet)|$.

– Par un raisonnement similaire, si $p\bullet \in \mathcal{N}_s$, alors pour tout i :

$$l+i \curvearrowright_q \circ \curvearrowright_{p\bullet}^{(b)} \text{idx}_b(R(p\bullet), l+i) \quad \text{(B.11)}$$

Soit $i = k'|R(p\bullet)|_1 + i'$ et $i' < |R(p\bullet)|_1$. On peut décomposer l'expression de la façon suivante :

$$\text{idx}_b(R(p\bullet), l+i) = \text{idx}_b(R(p\bullet), l) + k'|R(p\bullet)| + y \quad \text{(B.12)}$$

où y est la longueur du plus court suffixe de $R(p\bullet)$ allant jusqu'au $i'^{\text{ème}}$ «b». Puis nous posons :

$$x = \text{idx}_b(R(p\bullet), l) + k'|R(p\bullet)| + y + M(q) \quad \text{(B.13)}$$

$$y = x + \frac{|R(p\bullet)|}{|R(p\bullet)|_b} kj \quad \text{(B.14)}$$

L'équivalence est vérifiée par récurrence pour tout k, et pour tout j de la forme $k''|R(p\bullet)|_b$.

Par conséquent, posons :

$$j = \prod_{n \in \mathcal{N}_s \cup \mathcal{N}_f} \max(1, |R(n)| \times |R(n)|_0 \times |R(n)|_1) \quad \text{(B.15)}$$

Ainsi, j est un multiple de tous $|R(n)|$, $|R(n)|_0$ et $|R(n)|_1$. j est également un multiple de la période du graphe (plus petit commun multiple des périodes des nœuds). Lorsque j jetons ont été consommés en place p, αj jetons ont été produits en place q, où, selon la nature de $p\bullet$, α vaut soit 1 (calcul ou copie), soit $\frac{|R(p\bullet)|_b}{|R(p\bullet)|}$ (sélection), soit $\frac{|R(p\bullet)|}{|R(p\bullet)|_b}$ (fusion). Finalement, en s'appuyant sur le fait que les routages sont périodiques, l'équivalence $p_{l+i} \overset{f}{\sim} p_{l+i+kj} \Leftrightarrow q_x \overset{f}{\sim} q_y$ est vraie par récurrence. □

Preuve de la proposition 3.25. Vérifions que la relation $\precsim$ vérifie les deux propriétés d'une relation d'équivalence, à savoir la réflexivité et la transitivité.

– *Réflexivité* : $p_i = p_i$, donc $p_i \precsim p_i$
– *Transitivité* : par définition, $\precsim$ est une clôture transitive, elle est évidemment transitive.

□

Preuve de la proposition 3.28. Prouvons que la relation $\overset{b}{\sim}$ vérifie les trois propriétés d'une relation d'équivalence, à savoir la réflexivité, la symétrie et la transitivité.

– *Réflexivité* : la première moitié de la relation est réflexive, car $p_i \overset{f}{\sim} p_i$ par la proposition 3.17. L'autre moitié est vraie également, puisque $p = p$.
– *Symétrie* : $\overset{f}{\sim}$ est symétrique par la proposition 3.17. L'autre moitié de la proposition est évidemment symétrique : l'égalité est transitive, et il n'existe pas de tel q_j.
– *Transitivité* : la relation $\overset{b}{\sim}$ est, par définition, une clôture transitive.

□

Preuve de la proposition 3.29. Cette proposition est directement induite par la définition 3.27. □

Preuve du théorème 3.31. Dans les sous-sections 3.2.2 à 3.2.5, il a été montré comment extraire des dépendances de données d'un KRG selon sa topologie, son routage et son marquage initial ; et comment les représenter sous la forme d'un $\overset{b}{\sim}$-GDR. Cette méthode s'applique à un quelconque KRG, donc à tout KRG correspond un unique $\overset{b}{\sim}$-GDR. La question consiste maintenant à savoir si l'algorithme 3.1 permet effectivement de construire un quasi-graphe marqué respectant la sémantique du $\overset{b}{\sim}$-GDR.

Pour cela, il faut montrer point par point qu'il existe une bijection entre les deux graphes : construire un $\overset{b}{\sim}$-GDR $g' = \langle \mathcal{M}', \mathcal{D}', \mathcal{I}', \mathcal{Y}' \rangle$ à partir du quasi-graphe marqué, et le comparer avec le $\overset{b}{\sim}$-GDR initial $g = \langle \mathcal{M}, \mathcal{D}, \mathcal{I}, \mathcal{Y} \rangle$. Nous ne donnerons ici qu'un résumé de la preuve, dont les éléments sont les suivants :

Nœuds du quasi-graphe marqué : Tout nœud a au moins un arc entrant, d'où $\mathcal{I}' = \mathcal{Y}'$.

Jetons initiaux et productibles : Le quasi-graphe marqué contient au plus un jeton par place. Par ailleurs, le quasi-graphe marqué contient autant de jetons qu'il y en a dans $\mathcal{Y}$. On a alors $\mathcal{Y} = \mathcal{Y}'$.

Places du quasi-graphe marqué : À tout jeton de $\mathcal{M}$ correspond une place dans le quasi-graphe marqué. L'algorithme ne crée des places supplémentaires que dans trois cas :
- Circuit sans marquage (*cf.* FIG. 3.12(d)) : c'est une dépendance circulaire sans donnée initiale que l'on peut éliminer à la construction de g' ;
- Copies et fusions (*cf.* FIG. 3.12(f) et (j)) : les jetons qui y transitent sont abstraits par équivalence à la construction de g'.
- Lier les copies aux nœuds de calcul et de fusion (étape 5) : les jetons sont également abstraits par équivalence à la construction de g'.

Dépendances : L'étape 5 de l'algorithme crée un lien pour chaque dépendance de $\mathcal{D}$. On peut vérifier que par construction, tout circuit du quasi-graphe marqué contient un marquage au plus unitaire. La correspondance d'un jeton de $\mathcal{M}$ pour chaque place (autre que celles mentionnées au point précédent) est conservée par équivalence, et $\mathcal{M} = \mathcal{M}'$. On montre alors que, par construction, les dépendances le sont également.

g et g' ont donc les mêmes jetons (à un renommage près), les mêmes jetons productibles, et les mêmes dépendances. La seule différence concerne les ensembles de jetons initiaux : $\mathcal{I} \subseteq \mathcal{Y}$ tandis que $\mathcal{I}' = \mathcal{Y}'$. Ce point est discuté en remarque 3.32. Cependant, l'algorithme ne prend pas en compte les jetons initiaux pour la construction d'un quasi-graphe marqué ; seuls les jetons productibles comptent. Ainsi, les quasi-graphes marqués construits à partir de g et g' sont égaux : il y a bien unicité de la forme normale. □

Preuve du théorème 3.33. Comme nous l'avons vu, un $\overset{f}{\sim}$-quasi-graphe marqué préserve les dépendances de flots, mais peut fusionner des calculs sans tenir compte de la séquentialité des jetons, contrairement au $\overset{b}{\sim}$-quasi-graphe marqué. C'est donc cette seconde forme normale que nous considérons afin d'établir la vivacité du modèle.

Par définition, un KRG ou un quasi-graphe marqué sont quasi-vivants si l'un de leur nœud peut être tiré à l'infini. De plus, il est vivant si tout nœud peut être tiré à l'infini. Cette propriété peut-être transposée dans le domaine des dépendances de données : un KRG ou un quasi-graphe marqué sont quasi-vivants si les dépendances de l'un de leurs flots de jetons sont satisfaites à l'infini.

Aussi, un KRG et son $\overset{b}{\sim}$-quasi-graphe marqué sont liés par un même $\overset{b}{\sim}$-GDR : ainsi, si les dépendances sont satisfaites dans l'un, elle le sont dans le second. □

Preuve du lemme 3.34. On illustre la preuve par l'exemple de la FIG. 3.13 (a) *(gauche)*. On suppose par le membre gauche de l'implication que le $[u]_i^{ème}$ jeton ayant franchi la fusion est routé sur la sortie «1» de la sélection ; en d'autres termes, le $i^{ème}$ jeton du flot b sort dans le flot d. On montre que les deux membres de l'égalité sont deux façons distinctes d'écrire l'indice dans le flot d de ce $i^{ème}$ jeton de b.

Du côté gauche de l'égalité : $[u]_i$ représente la position, dans le flot de jetons ayant franchi la fusion, du $i^{ème}$ jeton de b (position du $i^{ème}$ «1» dans la séquence de routage). $\text{pre}\,(v,[u]_i)$ correspond au routage de ces i jetons sur le nœud de sélection. Le nombre de «1» qu'il contient est donc égal à l'indice attribué au dernier de ces jetons produits dans d.

Du côté droit de l'égalité : $v\blacktriangledown u$ est un échantillonnage de la séquence de routage de la sélection par celle de la fusion : c'est donc le routage sur la sélection des jetons arrivés par b. Ainsi, $|\text{pre}\,(v\blacktriangledown u,i)|_1$ est le nombre d'entre eux qui, jusqu'au $i^{ème}$, sont aiguillés sur d. D'autre part, $u\blacktriangledown v$ est l'entrée d'origine des jetons sortant de la sélection par d. Finalement, $[u\blacktriangledown v]_{|\text{pre}(v\blacktriangledown u,i)|_1}$ est la position, parmi tous les jetons de d, du $i^{ème}$ à suivre le chemin (b,d).

Ces deux notations sont bien deux manières différentes de désigner un même jeton ; elles sont donc équivalentes. □

Preuve de la proposition 3.35. Établissons les relations entre les flots d'entrée et de sortie, en composant les relations de flot :

FIG. 3.13(a) *(gauche)* :

$$a \curvearrowright_s^{(0)} \circ \curvearrowright_p \circ \curvearrowright_m^{(0)} |\text{prf}\,(\overline{v},\text{idx}_1\,(\overline{u},a))|_1 \tag{B.16}$$
$$a \curvearrowright_s^{(1)} \circ \curvearrowright_p \circ \curvearrowright_m^{(0)} |\text{prf}\,(v,\text{idx}_1\,(\overline{u},a))|_1 \tag{B.17}$$
$$b \curvearrowright_s^{(0)} \circ \curvearrowright_p \circ \curvearrowright_m^{(1)} |\text{prf}\,(\overline{v},\text{idx}_1\,(u,b))|_1 \tag{B.18}$$
$$b \curvearrowright_s^{(1)} \circ \curvearrowright_p \circ \curvearrowright_m^{(1)} |\text{prf}\,(v,\text{idx}_1\,(u,b))|_1 \tag{B.19}$$

FIG. 3.13(a) *(droite)* :

$$a \curvearrowright_m^{(0)} \circ \curvearrowright_p \circ \curvearrowright_s^{(0)} \text{idx}_1\,(\overline{u}\triangle\overline{v},|\text{prf}\,(\overline{v}\triangle\overline{u},a)|_1) \tag{B.20}$$
$$a \curvearrowright_m^{(0)} \circ \curvearrowright_p \circ \curvearrowright_s^{(1)} \text{idx}_1\,(\overline{u}\triangle v,|\text{prf}\,(v\triangle\overline{u},a)|_1) \tag{B.21}$$
$$b \curvearrowright_m^{(1)} \circ \curvearrowright_p \circ \curvearrowright_s^{(0)} \text{idx}_1\,(u\triangle\overline{v},|\text{prf}\,(\overline{v}\triangle u,b)|_1) \tag{B.22}$$
$$b \curvearrowright_m^{(1)} \circ \curvearrowright_p \circ \curvearrowright_s^{(1)} \text{idx}_1\,(u\triangle v,|\text{prf}\,(v\triangle u,b)|_1) \tag{B.23}$$

On montre ensuite que les relations sont égales deux à deux, par application directe du lemme 3.34. □

Preuve de la proposition 3.37. Nous établissons les relations entre flots d'entrée et de sortie, par les définitions 3.5 et 3.6. Nous avons :

FIG. 3.15(a) *(gauche)* :

$$a \curvearrowright_m^{(0)} \text{idx}_1\,(\overline{v},a) \tag{B.24}$$
$$b \curvearrowright_m^{(1)} \circ \curvearrowright_p \circ \curvearrowright_m^{(0)} \text{idx}_1\,(v,\text{idx}_1\,(\overline{u},b)) \tag{B.25}$$
$$c \curvearrowright_m^{(1)} \circ \curvearrowright_p \circ \curvearrowright_m^{(1)} \text{idx}_1\,(v,\text{idx}_1\,(u,c)) \tag{B.26}$$

FIG. 3.15(a) *(droite)* :

$$a \curvearrowright_m^{(0)} \circ \curvearrowright_p \circ \curvearrowright_m^{(0)} \mathrm{idx}_1\left(\overline{v \blacktriangledown u}, \mathrm{idx}_1\left(\overline{v \triangle \overline{v \blacktriangledown u}}, a\right)\right) \tag{B.27}$$

$$b \curvearrowright_m^{(0)} \circ \curvearrowright_p \circ \curvearrowright_m^{(1)} \mathrm{idx}_1\left(\overline{v \blacktriangledown u}, \mathrm{idx}_1\left(v \triangle \overline{v \blacktriangledown u}, b\right)\right) \tag{B.28}$$

$$c \curvearrowright_m^{(1)} \mathrm{idx}_1\left(v \blacktriangledown u, c\right) \tag{B.29}$$

Puis par le corollaire A.23 :
(a)

$$\begin{aligned}
\overline{v \blacktriangledown u} \blacktriangledown v \triangle \overline{v \blacktriangledown u} &= (v \blacktriangledown u) \oplus \overline{\overline{v \blacktriangledown u} \blacktriangledown \left(v \triangle \overline{v \blacktriangledown u}\right)} \quad \text{par prop. A.29} \\
&= (v \blacktriangledown u) \oplus \overline{v \wedge \overline{v \blacktriangledown u}} \quad \text{par prop. A.41} \\
&= (v \blacktriangledown u) \oplus (\overline{v} \vee (v \blacktriangledown u)) \\
&= \overline{v}
\end{aligned}$$

(b)

$$\begin{aligned}
\overline{v \blacktriangledown u} \blacktriangledown \left(v \triangle \overline{v \blacktriangledown u}\right) &= v \wedge \overline{v \blacktriangledown u} \quad \text{par prop. A.41} \\
&= v \wedge (\overline{v} \oplus (v \blacktriangledown \overline{u})) \\
&= v \blacktriangledown \overline{u}
\end{aligned}$$

(c)

$$v \blacktriangledown u = v \blacktriangledown u$$

Par identification, nous constatons que les relations sur les flots sont égales deux à deux. □

Preuve de la proposition 3.38. De même que précédemment, nous établissons les relations entre flots d'entrée et de sortie :
FIG. 3.15(b) *(gauche)* :

$$\mathrm{idx}_1\left(\overline{u}, b\right) \curvearrowright_s^{(0)} b \tag{B.30}$$

$$\mathrm{idx}_1\left(u, \mathrm{idx}_1\left(\overline{v}, c\right)\right) \curvearrowright_s^{(0)} \circ \curvearrowright_p \circ \curvearrowright_s^{(1)} c \tag{B.31}$$

$$\mathrm{idx}_1\left(u, \mathrm{idx}_1\left(v, d\right)\right) \curvearrowright_s^{(1)} \circ \curvearrowright_p \circ \curvearrowright_s^{(1)} d \tag{B.32}$$

FIG. 3.15(b) *(droite)* :

$$\mathrm{idx}_1\left(\overline{u \blacktriangledown v}, \mathrm{idx}_1\left(\overline{u \triangle \overline{u \blacktriangledown v}}, b\right)\right) \curvearrowright_s^{(0)} \circ \curvearrowright_p \circ \curvearrowright_s^{(0)} b \tag{B.33}$$

$$\mathrm{idx}_1\left(\overline{u \blacktriangledown v}, \mathrm{idx}_1\left(u \triangle \overline{u \blacktriangledown v}, c\right)\right) \curvearrowright_s^{(1)} \circ \curvearrowright_p \circ \curvearrowright_s^{(0)} c \tag{B.34}$$

$$\mathrm{idx}_1\left(u \blacktriangledown v, d\right) \curvearrowright_s^{(1)} d \tag{B.35}$$

La suite de la preuve est identique à celle de la la proposition 3.37, à un renommage près : u devient v et v devient u. □

Preuve du corollaire 3.39. Par les propositions 3.35, 3.37 et 3.38, ces transformations préservent les flots de jetons. Par conséquent, les transformations sont locales et ne modifient pas le comportement global du KRG. □

Preuve du théorème 3.46. Donnée par Adelson-Velskii et Landis [1] et Knuth [135]. □

Preuve de la proposition 3.51. Corollaire 7.4 de Golumbic [101]. □

Preuve de la proposition 3.53. La coloration du graphe nous donne l'allocation des jetons à chaque canal du trieur. Ces canaux sont numérotés de 0 à $X-1$ (soit X canaux) ; ils sont connectés aux sorties de la X-sélection et aux entrées de la X-fusion, elles-mêmes numérotées de 0 à $X-1$, de telle façon que le $i^{\text{ème}}$ canal reçoit les jetons issus de la $i^{\text{ème}}$ sortie de la sélection, et les dirige vers la $i^{\text{ème}}$ entrée de la fusion. Ainsi, si le $k^{\text{ème}}$ jeton est produit (et consommé) dans la place $\chi(k)$.

La suite de la preuve se base sur une double récurrence : d'une part sur X, d'autre part sur la longueur de la permutation π. Si $X=1$, alors π est la permutation *identité*, et $R(s)=R(f)$; le cas est trivial. Si $X>1$, et si la longueur de π vaut 1, alors π est encore une fois l'identité : tous les jetons suivent le même chemin, il n'y a pas de dépassement. Enfin, dans le cas où $X>1$, par une récurrence sur X, puis sur la longueur de π, on montre que les équations (3.21) et (3.22) nous permettent d'effectuer la permutation π sur les X flots de jetons. □

Preuve du théorème 3.54. Considérons deux nœuds de calcul n et n', tels qu'il existe un chemin σ allant de l'un vers l'autre, via des places, et nœuds de fusion et de sélection (ni nœuds de calcul, ni de copies).

Tout d'abord, nous appliquons la transformation *ExpPlace* sur σ tant que des nœuds de fusion sont connectés par leurs sorties aux entrées de nœuds de sélection. Ainsi, nous «poussons» les nœuds de fusion «vers le haut», tandis que les nœuds de sélection sont «poussés vers le bas». Par ailleurs, nous éliminons les boucles formées par des nœuds de sélection et de fusion : appliquer récursivement la transformation *ExpPlace* revient à dérouler entièrement la boucle. Nous répétons la transformation à tout chemin σ entre n et n', de telle façon qu'il traverse *d'abord* un arbre de sélection, *puis* un arbre de fusion.

La seconde étape consiste à réordonner les arbres de sélection et de fusion. Nous appliquons les transformations *PermFusion* et *PermSelect* afin de «décroiser» les arcs, tout en nous appuyant sur les remarques 3.40 et 3.41 pour effectuer les simplifications. Nous répétons le processus jusqu'à obtenir un trieur parallèle.

Le nombre de chemins du trieur ainsi obtenu n'est pas nécessairement minimal (*cf.* sous-section 3.3.3). En observant le routage des jetons au cours des phases initiale et périodique, nous en déduisons la permutation effectuée. Par la remarque 3.40, nous décomposons le flot de jetons en autant de séquences de jetons permutées, et les affectons à des places différentes ; puis nous réordonnons à nouveau les sélections et fusions par applications successives des transformations *PermFusion* et *PermSelect*. Finalement, nous fusionnons les places dont les jetons ne sont pas permutés, par la remarque 3.40.

Nous répétons ces transformations pour tout couple de nœuds de calcul n et n'. La topologie du graphe ainsi obtenu est d'une forme similaire à celle de la FIG. 3.25. □

Preuve du lemme 3.66. L'inéquation (3.26) correspond à la règle d'activation du nœud, tandis que l'équation (3.27) correspond à la règle de tirage. Dans les deux cas, ces contraintes sont aussi liées aux relations de flots (*cf.* définition 3.5).

Dans le premier cas (inéquation (3.26)), le $i^{\text{ème}}$ jeton sortant de la place p est en relation avec le $i^{\text{ème}}$ jeton consommé par n. Par transitivité, le $i^{\text{ème}}$ jeton entrant dans p est donc en relation avec le $(M(p)+1)^{\text{ème}}$ jeton consommé par n. À cause des latences, ce jeton ne sera prêt à être consommé dans p qu'au moins $L(p)$ instants après y avoir été produit. La règle d'activation ne donne qu'une borne

minimum sur la distance entre les instants ; on en conclut la relation entre leurs ordonnancements, sous forme d'inéquation.

Dans le second cas (équation (3.27)), le $i^{\text{ème}}$ jeton produit par n est en relation avec le $i^{\text{ème}}$ jeton produit dans p. De plus, la règle de tirage impose une contrainte forte sur les ordonnancements : un jeton est produit dans la place en sortie exactement $L(n)$ instants après le tirage de n. On en déduit l'équation sur leurs ordonnancements.

Notons que l'inéquation (3.26) est valable pour toute entrée de n ; par conséquent, la $i^{\text{ème}}$ exécution de n est telle que :

$$\max_{p \in n\bullet} t_p \leq \text{idx}_1(T(n), i) \tag{B.36}$$

$$t_p = \begin{cases} \text{idx}_1(T(p), i - M(p)) + L(p) & \text{si } i - M(p) > 0 \\ 0 & \text{sinon} \end{cases} \tag{B.37}$$

□

Preuve du lemme 3.67. De même que dans le lemme précédent, les contraintes dépendent à la fois des relations sur les flots de jetons, et des règles d'activation et de tirage. En l'occurrence, les inéquations (3.30) et (3.31) d'une part, et l'équation (3.32) d'autre part, incarnent respectivement les contraintes d'activation et de tirage.

Les inéquations (3.30) et (3.31) correspondent aux contraintes d'activation selon que les jetons sont consommés sur les entrées «0» ou «1» du nœud, respectivement. C'est la raison pour laquelle l'ordonnancement de n est filtré par sa séquence de routage, afin de déterminer quels tirages correspondent à une consommation dans l'une ou l'autre des places : par exemple, $T(n) \blacktriangledown R(n)$ pour $\bullet_1 n$. Comme dans le lemme 3.66, chacun de ces ordonnancements est contraint par les arrivées des jetons dans les places en entrée, d'où les inéquations.

Nous savons aussi que les nœuds de fusion ont des latences nulles. Ainsi, lorsque le nœud est tiré, il produit simultanément un jeton dans la place en sortie ; les ordonnancement sont donc égaux, d'où l'équation (3.32). □

Preuve du lemme 3.68. Même principe que les preuves des lemmes 3.66 et 3.67. □

Preuve du théorème 3.69. Comme mentionné précédemment, les équations de balance augmentées doivent permettre de vérifier deux propriétés fondamentales :

1. L'équilibre entre productions et consommations est la propriété vérifiée par les équations de balance de SDF. Ici, elles s'écrivent sous la forme de l'équation (3.42) : pour toute place, le producteur et le consommateur sont synchronisables. Cela signifie que les jetons sont produits et consommés à des débits égaux (*cf.* proposition A.13), et que les places ont des capacités bornées (*cf.* proposition A.15).
2. Le respect de la causalité et des règles d'activation et de tirage est introduit dans ce théorème par le second point. C'est une application directe des lemmes 3.66, 3.67 et 3.68.

□

Preuve de la proposition 3.70. La capacité d'une place doit être suffisante pour contenir la différence de jetons entre la production en amont et la consommation en aval, plus le nombre de jetons initialement présents. Par définition, une place p contient initialement $M(p)$ jetons. La différence entre jetons produits et consommés évolue, elle, au cours du temps. Par les règles d'activation et de tirage, nous

savons que cette différence ne peut être inférieure à $-M(p)$. Par ailleurs, nous considérons un SKRG borné, donc cette différence est bornée au cours du temps.

Si les producteur et consommateur ne sont pas des nœuds de routage, alors la place reçoit un nouveau jeton à chaque exécution du producteur n_1, et un jeton est consommé à chaque exécution du consommateur n_2. Au $t^{\text{ème}}$ instant, le compte de jetons dans p a donc été incrémenté de $|\text{pre}(T(n_1),t)|_1$, et décrémenté de $|\text{pre}(T(n_2),t)|_1$.

En revanche, si n_1 est un nœud de sélection, alors la production dans p est «filtrée» par la séquence de routage $R(n_1)$; le compte de jetons dans p sera alors incrémenté de $|\text{pre}(R(n_1),|\text{pre}(T(n_1),t)|_1)|_{b_1}$. Le principe est le même si n_2 est un nœud de fusion, d'où les valeurs de α et β.

L'ordonnancement du SKRG étant k-périodique, il ne reste plus qu'à prendre la différence maximale au cours de la phase d'initialisation et la première période (borne sur t). □

Preuve de la proposition 3.74. On cherche à minimiser la capacité totale du mélangeur, c'est-à-dire la somme des capacités des places. On impose donc les contraintes suivantes :

- D'après leurs définitions, $\chi_{i,k}$ est binaire, et la capacité d'une place est un entier naturel.
- $\sum_{k=1}^{X}\chi_{i,k}=1$ impose qu'à chaque nœud i est associée une et une seule couleur k, donc qu'un jeton ne passe que par une et une seule place.
- $(\chi_{i,k}+\chi_{j,k})P_{i,j}\leq 1$ impose que parmi deux nœuds adjacents dans le graphe de permutation, au plus un et de couleur k, donc que deux jetons permutés ne transitent pas par le même chemin.

Jusqu'alors, les contraintes sont celles de Grund-Hack ou Appel-George. La dernière contrainte donne la borne minimale de la capacité c_k d'une place, en fonction de la coloration des nœuds (routage des jetons).

Tout d'abord, le produit de la parenthèse par $\chi_{i,k}$ permet de ne s'intéresser qu'au cas où le nœud i est de couleur k, l'expression valant zéro sinon. Voyons maintenant le détail de cette parenthèse. La somme des $\chi_{j,k}I_{j,i}$ nous donne le nombre de jetons présents dans la place k lorsque le jeton i arrive $(j<i)$, augmenté d'un si ce $i^{\text{ème}}$ jeton ne sort pas dans le même instant $(+1-C_i)$. On a alors le nombre total de jetons stockés dans la place à cet instant, $i^{\text{ème}}$ inclus. □

Preuve de la proposition 3.75. Étudions le comportement du trieur parallèle, instant après instant, dans les différents cas de figure. Il a quatre états :

1. *aucun jeton ne sort* : soit le mélangeur est vide, soit tous les jetons présents, y compris celui venant éventuellement d'y entrer, sont arrêtés dans l'attente d'être dépassés par l'un de leurs successeurs. Au prochain instant, nous restons en (1) ou allons en (2).
2. *un jeton entre et sort* : le jeton entrant a doublé tous les autres pour ressortir aussitôt. Ensuite, nous restons en (2) ou allons en (1), (3) ou (4).
3. *un jeton entre, un autre sort* : le jeton entrant a été stocké et un jeton plus prioritaire est sorti. À l'instant suivant, nous restons en (3) ou allons en (1), (2) ou (4).
4. *un jeton sort, aucun n'entre* : c'est le seul état dans lequel la permutation peut se terminer. Sinon, nous restons en (4) ou allons en (1), (2) ou (3).

Pour résumer, le seul cas dans lequel aucun jeton ne sort du trieur est celui d'attente d'un successeur, ce qui dépend uniquement de l'ordonnancement des jetons en entrée, et non de la topologie du trieur. Dans tous les autres cas, le trieur peut produire un jeton en sortie, par des tirages *au plus tôt*. Aucun retard n'est donc introduit par le trieur ; le tri est optimal en temps. □

B.3 Preuves du chapitre 4

Démonstration. Une première preuve du lemme est donnée dans [83]. Voir Feautrier [86] et Gale *et al.* [91] pour plus de détails. □

B.4 Preuves de l'annexe A

Preuve de la proposition A.6. Par application immédiate de la définition A.5. □

Preuve de la proposition A.12. Proposition 2 de Cohen *et al.* [64]. □

Preuve de la proposition A.13. Proposition 6 de Cohen *et al.* [63]. □

Preuve de la proposition A.14.
($\Rightarrow$) Supposons que cette limite soit différente de 0. Cela équivaut à dire que $\lim_{i\to+\infty} |\mathrm{prf}(u,i)|_1 - |\mathrm{prf}(v,i)|_1 = +\infty$ (nous pouvons interchanger u et v pour éliminer le cas négatif). Si v est de la forme $w.0^\omega$ (avec $w \in \mathbb{B}^*$), il est évident que u et v ne sont pas synchronisables. Dans le cas contraire, pour i tendant vers l'infini, on peut écrire le préfixe de v contenant $n_i = |\mathrm{prf}(u,i)|_1$ «1» sous la forme $\mathrm{prf}(v,i).w_i$, où w_i est un sous-mot de v dépendant de i. Puisque $\lim_{i\to+\infty} |w_i|_1 = \lim_{i\to+\infty} (n_i - |\mathrm{prf}(v,i)|_1) = +\infty$, on a $\lim_{i\to+\infty} |w_i| = +\infty$. Par conséquent, la distance $|w_i|$ entre les $n_i^{ème}$ «1» de u et de v n'est pas bornée, donc u et v ne sont pas synchronisables (proposition A.12).

($\Leftarrow$) Si u et v sont synchronisables, nous avons par la proposition A.12 que pour tout i, les distances entre les $i^{ème}$ «1» de u et v sont bornées par d_1 et d_2. L'un des pires cas est donc celui où les lettres de u, de la $(i+1)^{ème}$ à la $(i+d_2)^{ème}$, ne sont que des «1». Nous avons alors $|\mathrm{prf}(u,i)|_1 + d_2 \geq |\mathrm{prf}(v,i)|_1$, donc $|\mathrm{prf}(v,i)|_1 - |\mathrm{prf}(u,i)|_1 \leq d_2$. À l'inverse, l'autre extremum est $|\mathrm{prf}(u,i)|_1 - |\mathrm{prf}(v,i)|_1 \leq d_1$, dans le cas où les lettres de v, de la $(i+1)^{ème}$ à la $(i+d_1)^{ème}$, ne seraient que des «1». La différence du nombre de «1» dans les préfixes de longueur i étant bornée, nous en concluons que la limite du quotient par i tend vers 0 au voisinage de l'infini. □

Preuve de la proposition A.15. Par application directe de la proposition A.14 (*cf.* sa preuve). □

Preuve de la proposition A.20. L'équation A.22 est donnée directement par la définition A.16 : pour toute lettre de u, une lettre est concaténée au résultat, et $|u\blacktriangledown v| = |u|$.

L'équation A.23 vient du fait que chaque «1» de u est remplacé par une lettre de v. Ainsi, le résultat contient autant que «1» que v, et $|u\blacktriangledown v|_1 = |v|_1$.

Enfin, l'équation A.24 est inférée des deux précédentes :

$$\begin{aligned}
|u\blacktriangledown v|_0 &= |u\blacktriangledown v| - |u\blacktriangledown v|_1 \\
&= |u| - |v|_1 \\
&= |u|_0 + |u|_1 - |v|_1 \\
&= |u|_0 + |v| - |v|_1 \\
&= |u|_0 + |v|_0 + |v|_1 - |v|_1 \\
&= |u|_0 + |v|_0
\end{aligned}$$

□

Preuve de la proposition A.21.

$$\text{débit}\,(u\blacktriangledown v) = \frac{|u\blacktriangledown v|_1}{|u\blacktriangledown v|} = \frac{|v|_1}{|u|} = \frac{|v|}{|u|} \times \frac{|v|_1}{|v|} = \frac{|u|_1}{|u|} \times \frac{|v|_1}{|v|} = \text{débit}\,(u) \times \text{débit}\,(v)$$

□

Preuve de la proposition A.22. Par définition, chaque «1» de u est remplacé par la lettre correspondante de v. Nous avons $\forall\, i \in [\![1, |u|]\!],\ (u\blacktriangledown v)_i \Rightarrow u_i$. D'où $u\blacktriangledown v \sqsubseteq u$. □

Preuve du corollaire A.23. Par application directe des définitions A.5 et A.16 :

$$u\blacktriangledown v \sqsubseteq u \ \Leftrightarrow\ \forall\, i \in \mathbb{N},\ \exists\, j \in \mathbb{N},\ \text{idx}_1\,(u, j) = \text{idx}_1\,(u\blacktriangledown v, i)$$

où $j = \text{idx}_1\,(v, i)$. □

Preuve de la proposition A.24. D'une part, si chaque «1» de $1^{|u|}$ est remplacé par la lettre correspondante de u, le résultat est égal à u. D'autre part, si chaque «1» de u est remplacé par un «1» de $1^{|u|_1}$, le résultat est lui aussi égal à u. □

Preuve de la proposition A.25. Par définition, la longueur de l'opérande droit est égale au nombre de «1» dans l'opérande gauche. Les seuls mots vérifiant cette propriété sont ceux composés uniquement de «1» (et ε). □

Preuve de la proposition A.26. Donnée par Cohen *et al.* [63]. □

Preuve de la proposition A.27. On montre par récurrence la correction de l'égalité sur la conjonction. Pour $|u| = 0$:

$$\varepsilon\blacktriangledown\,(\varepsilon \wedge \varepsilon) = \varepsilon = (\varepsilon\blacktriangledown\varepsilon) \wedge (\varepsilon\blacktriangledown\varepsilon)$$

Puis on suppose l'égalité vraie pour $|u| = n-1$, et on montre la correction pour $|u| = n$. Si $u_{tête} = 0$,

$$\begin{aligned} u\blacktriangledown\,(v \wedge w) &= 0.\,(u_{queue}\blacktriangledown\,(v_{queue} \wedge w_{queue})) \\ &= 0.\,((u_{queue}\blacktriangledown v_{queue}) \wedge (u_{queue}\blacktriangledown w_{queue})) \\ &= 0.\,(u_{queue}\blacktriangledown v_{queue}) \wedge 0.\,(u_{queue}\blacktriangledown w_{queue}) \\ &= (u\blacktriangledown v) \wedge (u\blacktriangledown w) \end{aligned}$$

Si $u_{tête} = 1$,

$$\begin{aligned} u\blacktriangledown\,(v \wedge w) &= (v_{tête} \wedge w_{tête})\,.\,(u_{queue}\blacktriangledown\,(v_{queue} \wedge w_{queue})) \\ &= (v_{tête} \wedge w_{tête})\,.\,((u_{queue}\blacktriangledown v_{queue}) \wedge (u_{queue}\blacktriangledown w_{queue})) \\ &= v_{tête}.\,(u_{queue}\blacktriangledown v_{queue}) \wedge w_{tête}.\,(u_{queue}\blacktriangledown w_{queue}) \\ &= (u\blacktriangledown v) \wedge (u\blacktriangledown w) \end{aligned}$$

Nous en concluons que la propriété est vraie quelque soit la longueur de u. Le même raisonnement par récurrence peut être utilisé pour prouver la correction des deux autres égalités. □

Preuve de la proposition A.28. ($\Rightarrow$)

$$\begin{aligned} u \sqsubseteq v &\Rightarrow (w\blacktriangledown u) \wedge (w\blacktriangledown v) = w\blacktriangledown (u \wedge v) \quad \text{par prop. A.27} \\ &\Rightarrow (w\blacktriangledown u) \wedge (w\blacktriangledown v) = w\blacktriangledown u \quad \text{par prop. A.6} \end{aligned}$$

$$\begin{aligned} u \sqsubseteq v &\Rightarrow (w\blacktriangledown u) \vee (w\blacktriangledown v) = w\blacktriangledown (u \vee v) \quad \text{par prop. A.27} \\ &\Rightarrow (w\blacktriangledown u) \vee (w\blacktriangledown v) = w\blacktriangledown u \quad \text{par prop. A.6} \end{aligned}$$

Toujours par la proposition A.6, on obtient :

$$u \sqsubseteq v \;\Rightarrow\; w\blacktriangledown u \sqsubseteq w\blacktriangledown v$$

($\Leftarrow$) $\forall i \in [\![1, |w|_1]\!]$,

$$\begin{aligned} w\blacktriangledown u \sqsubseteq w\blacktriangledown v &\Rightarrow (w\blacktriangledown u)_{\mathrm{idx}_1(w,i)} \Rightarrow (w\blacktriangledown v)_{\mathrm{idx}_1(w,i)} \\ &\Rightarrow u_i \Rightarrow v_i \\ &\Rightarrow u \sqsubseteq v \end{aligned}$$

□

Preuve de la proposition A.29.

$$\begin{aligned} & u = u\blacktriangledown 1^{|u|_1} \quad \text{par prop. A.24} \\ \Leftrightarrow\ & u = u\blacktriangledown (v \oplus \overline{v}) \\ \Leftrightarrow\ & u = (u\blacktriangledown v) \oplus (u\blacktriangledown \overline{v}) \quad \text{par prop. A.27} \\ \Leftrightarrow\ & u\blacktriangledown v = u \oplus (u\blacktriangledown \overline{v}) \\ \Leftrightarrow\ & \overline{u\blacktriangledown v} = \left(\overline{u} \wedge \overline{u\blacktriangledown \overline{v}}\right) \oplus (u \wedge (u\blacktriangledown \overline{v})) \\ \Leftrightarrow\ & \overline{u\blacktriangledown v} = \overline{u} \oplus (u\blacktriangledown \overline{v}) \quad \text{par prop. A.6 et A.22} \end{aligned}$$

La seconde égalité peut être prouvée de la même manière. □

Preuve du corollaire A.30. Soit $w = \overline{v}$. Par la proposition A.29 :

$$\begin{aligned} & \overline{u\blacktriangledown w} = \overline{u} \vee (u\blacktriangledown \overline{w}) \\ \Leftrightarrow\ & \overline{u\blacktriangledown \overline{v}} = \overline{u} \vee (u\blacktriangledown v) \\ \Leftrightarrow\ & \overline{\overline{u\blacktriangledown \overline{v}}} = \overline{\overline{u} \vee (u\blacktriangledown v)} \\ \Leftrightarrow\ & u\blacktriangledown \overline{v} = u \wedge \overline{u\blacktriangledown v} \end{aligned}$$

□

Preuve de la proposition A.31. Par application directe de la définition A.18,

$$u \triangle 1^{|u|} = w \;\Leftrightarrow\; \begin{cases} |w| = \left|1^{|u|}\right|_1 = |u| \\ \forall i \in [\![1, |u|]\!],\ w_i = u_i \end{cases}$$

puisque $\mathrm{idx}_1\left(1^{|u|}, i\right) = i$. Donc $w = u$. □

Preuve de la proposition A.32. Par définition, la longueur du résultat est égale au nombre de «1» dans l'opérande droit. Les seuls mots binaires vérifiant cette contrainte sont ceux composés uniquement de «1». □

Preuve de la proposition A.33. Par la définition A.18 et la proposition A.6, u échantillonne v. Chaque «1» dans u correspondant à un «1» dans v, il est évident que le résultat ne contient que des «1». □

Preuve de la proposition A.34.

$$\begin{aligned} u \sqsubseteq v \;\Rightarrow\; u = v\blacktriangledown(u\triangle v) &\Leftrightarrow\; u \sqsubseteq v \;\Rightarrow\; \text{débit}(u) = \text{débit}(v\blacktriangledown(u\triangle v)) \\ &\Leftrightarrow\; u \sqsubseteq v \;\Rightarrow\; \text{débit}(u) = \text{débit}(v) \times \text{débit}(u\triangle v) \\ &\Leftrightarrow\; u \sqsubseteq v \;\Rightarrow\; \text{débit}(u\triangle v) = \frac{\text{débit}(v)}{\text{débit}(u)} \end{aligned}$$

□

Preuve de la proposition A.35. Nous montrons par récurrence que l'égalité avec la conjonction est vraie. Tout d'abord, supposons que u, v et w sont de longueur 0. Nous avons :

$$(\varepsilon \wedge \varepsilon)\triangle\varepsilon = \varepsilon\triangle\varepsilon = \varepsilon = (\varepsilon\triangle\varepsilon) \wedge (\varepsilon\triangle\varepsilon)$$

Nous supposons maintenant l'égalité vraie pour des mots de longueur $n-1$ (les queues des mots binaires), et nous montrons sa correction pour des mots de longueur n. Si $w_{tête} = 0$,

$$\begin{aligned} (u \wedge v)\triangle w &= (u_{queue} \wedge v_{queue})\triangle w_{queue} \\ &= (u_{queue}\triangle w_{queue}) \wedge (v_{queue}\triangle w_{queue}) \\ &= (u\triangle w) \wedge (v\triangle w) \end{aligned}$$

Sinon si $w_{tête} = 1$,

$$\begin{aligned} (u \wedge v)\triangle w &= (u_{tête} \wedge v_{tête}).((u_{queue} \wedge v_{queue})\triangle w_{queue}) \\ &= (u_{tête}.(u_{queue}\triangle w_{queue})) \wedge (v_{tête}.(v_{queue}\triangle w_{queue})) \\ &= (u\triangle w) \wedge (v\triangle w) \end{aligned}$$

Donc l'égalité est vraie. Le même raisonnement par récurrence peut être suivi pour prouver la correction des égalités sur les disjonctions. □

Preuve de la proposition A.36.

$$\begin{aligned} u \sqsubseteq v &\Rightarrow (u\triangle w) \wedge (v\triangle w) = (u \wedge v)\triangle w \quad \text{par prop. A.35} \\ &\Rightarrow (u\triangle w) \wedge (v\triangle w) = u\triangle w \quad \text{par prop. A.6} \end{aligned}$$

$$\begin{aligned} u \sqsubseteq v &\Rightarrow (u\triangle w) \vee (v\triangle w) = (u \vee v)\triangle w \quad \text{par prop. A.35} \\ &\Rightarrow (u\triangle w) \vee (v\triangle w) = u\triangle w \quad \text{par prop. A.6} \end{aligned}$$

Enfin, toujours par la proposition A.6, nous obtenons :

$$u \sqsubseteq v \;\Leftrightarrow\; (u\triangle w) \sqsubseteq (v\triangle w)$$

□

Preuve de la proposition A.37.

$$\begin{array}{rll} & 1^{|v|_1} & = 1^{|v|} \triangle v \\ \Leftrightarrow & 1^{|v|_1} & = (u \oplus \overline{u}) \triangle v \\ \Leftrightarrow & 1^{|v|_1} & = (u \triangle v) \oplus (\overline{u} \triangle v) \quad \text{par prop. A.35} \\ \Leftrightarrow & u \triangle v & = 1^{|v|_1} \oplus \overline{u} \triangle v \\ \Leftrightarrow & \overline{u \triangle v} & = \left(1^{|v|_1} \wedge (\overline{u} \triangle v)\right) \vee \left(0^{|v|_1} \wedge \overline{\overline{u} \triangle v}\right) \\ \Leftrightarrow & \overline{u \triangle v} & = \overline{u} \triangle v \end{array}$$

□

Preuve de la proposition A.38. On montre par récurrence que l'égalité est vraie. Avec des mots de longueur nulle ($u = v = \varepsilon$) :

$$\varepsilon \wedge \varepsilon = \varepsilon \;\Rightarrow\; \varepsilon \triangle (\varepsilon \vee \varepsilon) = \overline{\varepsilon} \triangle (\varepsilon \vee \varepsilon)$$

Supposons désormais la propriété vraie avec des mots de longueur $n-1$, et prouvons la pour des mots de longueur n :

$$\begin{array}{rl} & u \wedge v = 0^{|u|} \;\Rightarrow\; u \triangle (u \vee v) = \overline{v} \triangle (u \vee v) \\ \Leftrightarrow & \left((u_{tête} \wedge v_{tête} = 0) \wedge \left(u_{queue} \wedge v_{queue} = 0^{|u_{queue}|}\right)\right. \\ \Rightarrow & (u_{tête} \triangle (u_{tête} \vee v_{tête})) \,.\, (u_{queue} \triangle (u_{queue} \vee v_{queue})) \\ = & (\overline{v_{tête}} \triangle (u_{tête} \vee v_{tête})) \,.\, (\overline{v_{queue}} \triangle (u_{queue} \vee v_{queue}))) \end{array}$$

Parce que nous avons supposé vrai :

$$u_{queue} \wedge v_{queue} = 0^{|u_{queue}|} \;\Rightarrow\; u_{queue} \triangle (u_{queue} \vee v_{queue}) = \overline{v_{queue}} \triangle (u_{queue} \vee v_{queue})$$

il reste à montrer :

$$u_{tête} \wedge v_{tête} = 0 \;\Rightarrow\; u_{tête} \triangle (u_{tête} \vee v_{tête}) = \overline{v_{tête}} \triangle (u_{tête} \vee v_{tête})$$

Si $u_{tête} \oplus v_{tête} = 1$, alors :

$$\begin{array}{rcl} u_{tête} \triangle (u_{tête} \vee v_{tête}) & = & u_{tête} \triangle 1 \\ & = & u_{tête} \\ & = & \overline{v_{tête}} \\ & = & \overline{v_{tête}} \triangle 1 \\ & = & \overline{v_{tête}} \triangle (u_{tête} \vee v_{tête}) \end{array}$$

Sinon si $u_{tête} \vee v_{tête} = 0$:

$$\begin{array}{rcl} u_{tête} \triangle (u_{tête} \vee v_{tête}) & = & u_{tête} \triangle 0 \\ & = & \varepsilon \\ & = & \overline{v_{tête}} \triangle 0 \\ & = & \overline{v_{tête}} \triangle (u_{tête} \vee v_{tête}) \end{array}$$

Ainsi l'égalité est vraie pour tous u and v. □

Preuve de la proposition A.39.

$$\begin{array}{rrcl} & v \sqsubseteq w & \Rightarrow & (u \wedge w) \triangle v = (u \triangle v) \wedge (w \triangle v) \quad \text{par prop. A.35} \\ \Leftrightarrow & v \sqsubseteq w & \Rightarrow & (u \wedge w) \triangle v = (u \triangle v) \wedge 1^{|w|_1} \\ \Leftrightarrow & v \sqsubseteq w & \Rightarrow & (u \wedge w) \triangle v = u \triangle v \end{array}$$

□

Preuve de la proposition A.40. Cette égalité est induite par les deuxièmes formulations des opérateurs *on* et *when*. Soit w un mot binaire tel que $w = v \blacktriangledown u$. Nous avons :

$$w = \begin{cases} w_{\mathrm{idx}_1(v,i)} = u_i & \forall\, i \in [\![1, |u|]\!] \\ w_i = 0 & \text{sinon} \end{cases}$$

qui est une autre façon d'exprimer que $u = w \triangle v$. D'où $u = (v \blacktriangledown u) \triangle v$. □

Preuve de la proposition A.41. La preuve est similaire à celle de la proposition précédente, avec cependant une différence : elle n'est vraie que si et seulement si u est un sous-flot de v, ce qui signifie qu'aucun «1» de u n'est omis lors de l'échantillonnage par v. Sinon, l'égalité $w_{\mathrm{idx}_1(v,i)} = u_i$ devient fausse. □

Preuve de la proposition A.42.

$$\begin{array}{rrcl} & v \blacktriangledown (u \triangle v) \blacktriangledown w & = & (u \triangle v) \blacktriangledown w \quad \text{par prop. A.41} \\ \Rightarrow & (v \blacktriangledown (u \triangle v) \blacktriangledown w) \triangle v & = & ((u \wedge v) \blacktriangledown w) \triangle v \quad \text{par prop. A.36} \\ \Leftrightarrow & (u \triangle v) \blacktriangledown w & = & ((u \wedge v) \blacktriangledown w) \triangle v \quad \text{par prop. A.40} \end{array}$$

□

Preuve de la proposition A.43. Nous montrons par récurrence la correction de l'égalité. Tout d'abord, nous supposons u de longueur nulle. Ainsi, $v = w = \varepsilon$.

$$(\varepsilon \blacktriangledown \varepsilon) \triangle (\varepsilon \blacktriangledown \varepsilon) = \varepsilon \triangle \varepsilon$$

Maintenant, nous supposons la propriété vraie pour u de longueur $n-1$ (la queue de u), et vérifions pour $|u| = n$. Si $u_{tête} = 0$, on a $\left|u_{queue}\right|_1 = |u|_1 = |v| = |w|$, et donc :

$$\begin{array}{rcl} (u \blacktriangledown v) \triangle (u \blacktriangledown w) & = & (0.u_{queue} \blacktriangledown v) \triangle (0.u_{queue} \blacktriangledown w) \\ & = & (0.(u_{queue} \blacktriangledown v)) \triangle (0.(u_{queue} \blacktriangledown w)) \\ & = & \varepsilon.((u_{queue} \blacktriangledown v) \triangle (u_{queue} \blacktriangledown w)) \\ & = & v \triangle w \end{array}$$

Sinon, si $u_{tête} = 1$, on a $\left|u_{queue}\right|_1 = |u|_1 - 1 = \left|v_{queue}\right| = \left|w_{queue}\right|$, d'où :

$$\begin{array}{rcl} (u \blacktriangledown v) \triangle (u \blacktriangledown w) & = & (1.u_{queue} \blacktriangledown v) \triangle (1.u_{queue} \blacktriangledown w) \\ & = & (v_{tête} \triangle w_{tête}).((u_{queue} \blacktriangledown v_{queue}) \triangle (u_{queue} \blacktriangledown w_{queue})) \\ & = & (v_{tête} \triangle w_{tête}).(v_{queue} \triangle w_{queue}) \\ & = & v \triangle w \end{array}$$

□

Preuve de la proposition A.44.

$$\begin{aligned} u\triangle(v\wedge w) &= (u\wedge w)\triangle(v\wedge w) \quad \text{par prop. A.39} \\ &= (w\blacktriangledown(u\triangle w))\triangle(w\blacktriangledown(v\triangle w)) \quad \text{par prop. A.41} \\ &= (u\triangle w)\triangle(v\triangle w) \quad \text{par prop. A.43} \end{aligned}$$

□

Preuve de la proposition A.45.

$$\begin{aligned} u\triangle(v\triangle w) &= (u\triangle(v\triangle w))\wedge 1^{|w|_1} \\ &= (u\triangle((v\wedge w)\triangle w))\wedge((v\triangle w)\triangle(v\triangle w)) \quad \text{par prop. A.36} \\ &= (u\wedge(v\triangle w))\triangle(v\triangle w) \quad \text{par prop. A.35} \\ &= (((w\blacktriangledown u)\triangle w)\wedge(v\triangle w))\triangle(v\triangle w) \quad \text{par prop. A.40} \\ &= (((w\blacktriangledown u)\wedge v)\triangle w)\triangle(v\triangle w) \quad \text{par prop. A.35} \\ &= (v\wedge(w\blacktriangledown u))\triangle(v\wedge w) \quad \text{par prop. A.44} \\ &= (w\blacktriangledown u)\triangle(v\wedge w) \quad \text{par prop. A.39} \end{aligned}$$

□

Preuve de la proposition A.46.

$$\begin{aligned} (u\blacktriangledown v)\wedge w &= (u\blacktriangledown v)\wedge(u\wedge w) \quad \text{par prop. A.6 et A.22} \\ &= (u\blacktriangledown v)\wedge(u\blacktriangledown(w\triangle u)) \quad \text{par prop. A.41} \\ &= u\blacktriangledown(v\wedge(w\triangle u)) \quad \text{par prop. A.27} \\ &= u\blacktriangledown((w\triangle u)\blacktriangledown(v\triangle(w\triangle u))) \quad \text{par prop. A.41} \\ &= (u\blacktriangledown(w\triangle u))\blacktriangledown(v\triangle(w\triangle u)) \quad \text{par prop. A.26} \\ &= (u\wedge w)\blacktriangledown(v\triangle(w\triangle u)) \quad \text{par prop. A.41} \end{aligned}$$

□

Preuve de la proposition A.47.

$$\begin{aligned} (u\blacktriangledown v)\triangle w &= ((u\blacktriangledown v)\wedge w)\triangle w \quad \text{par prop. A.39} \\ &= ((u\wedge w)\blacktriangledown(v\triangle(w\triangle u)))\triangle w \quad \text{par prop. A.46} \\ &= (u\triangle w)\blacktriangledown(v\triangle(w\triangle u)) \quad \text{par prop. A.42} \end{aligned}$$

□

Preuve de la proposition A.48.

$$\begin{aligned} u\triangle(v\blacktriangledown w) &= u\triangle(v\wedge(v\blacktriangledown w)) \\ &= (u\triangle v)\triangle((v\triangle w)\triangle v) \quad \text{par prop. A.44} \\ &= (u\triangle v)\triangle w \quad \text{par prop. A.40} \end{aligned}$$

□

Preuve de la proposition A.49.

$$\begin{aligned} u \triangle w = v \triangle w &\Rightarrow w \blacktriangledown (u \triangle w) = w \blacktriangledown (v \triangle w) \\ &\Rightarrow u \wedge w = v \wedge w \quad \text{par prop. A.41} \end{aligned}$$

□

Preuve du théorème A.59. Donnée par Clauss [61]. □

Bibliographie

[1] G. M. ADELSON-VELSKII et Y. M. LANDIS : An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962. Traduction anglaise par Myron J. Ricci dans Soviet Math. 3, p. 1259–1263.

[2] S. ADITYA GUPTA, B. R. RAU, V. K. KATHAIL et M. S. SCHLANSKER : Automatic design of VLIW processors. US Patent 6651222B2, novembre 2003.

[3] A. AGIWAL et M. SINGH : An architecture and a wrapper synthesis approach for multi-clock latency-insensitive systems. Dans *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'05)*, p. 1006–1013, Washington, DC, États-Unis, 2005. IEEE Computer Society.

[4] A. V. AHO, M. S. LAM, R. SETHI et J. D. ULLMAN : *Compilers : Principles, Techniques, and Tools*. Addison Wesley, seconde édition, août 2006.

[5] C. ALIAS, A. DARTE, P. FEAUTRIER et L. GONNORD : Multidimensional rankings, program termination, and complexity bounds of flowchart programs. Dans *Proceedings of the 17th International Static Analysis Symposium (SAS'10)*. Springer Verlag, septembre 2010.

[6] F. ANCEAU : A synchronous approach for clocking VLSI systems. *IEEE Journal of Solid-State Circuits*, 17:51–56, février 1982.

[7] C. ANDRÉ : Use of the behaviour equivalence in place-transition net analysis. Dans *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets*, p. 241–250, Londres, Royaume-Uni, 1982. Springer-Verlag.

[8] C. ANDRÉ : *Syntax and Semantics of the Clock Constraint Specification Language (CCSL)*. Rapport de recherche 6925, INRIA, juin 2009.

[9] C. ANDRÉ : *Verification of clock constraints : CCSL Observers in Esterel*. Rapport de recherche 7211, INRIA, février 2010.

[10] C. ANDRÉ, J. BOUCARON, A. COADOU, J. DEANTONI, B. FERRERO, F. MALLET et R. de SIMONE : MARTE/CCSL+TimeSquare+K-Passa : a design platform using formal MoCCs for embedded model-based engineering. Workshop SAFA'09, septembre 2009.

[11] A. W. APPEL et L. GEORGE : Optimal spilling for CISC machines with few registers. Dans *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*, p. 243–253. ACM, 2001.

[12] E. A. ASHCROFT et W. W. WADGE : Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[13] I. ATTALI, D. CAROMEL et A. L. WENDELBORN : From a formal dynamic semantics of Sisal to a Sisal environment. Dans *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, p. 266, Washington, DC, États-Unis, 1995. IEEE Computer Society.

[14] F. BACCELLI, G. COHEN, G. J. OLSDER et J.-P. QUADRAT : *Synchronization and Linearity*. John Wiley & Sons Ltd, Chichester, West Sussex, Royaume-Uni, 1992.

[15] J. W. BACKUS : Can programming be liberated from the Von Neumann style ? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[16] K. BANERJEE, S. IM et N. SRIVASTAVA : Interconnect modeling and analysis in the nanometer era : Cu and beyond. Dans *Proceedings of the 22nd Advanced Metallization Conference (AMC'05)*, p. 25–31. Materials Research Society, 2006.

[17] C. BASTOUL : Code generation in the polyhedral model is easier than you think. Dans *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, p. 7–16, Washington, DC, États-Unis, 2004. IEEE Computer Society.

[18] C. BASTOUL : *Improving Data Locality in Static Control Programs*. Thèse de doctorat, Université de Paris VI, décembre 2004.

[19] C. BASTOUL : *Extracting Polyhedral Representation from High Level Languages*. Rap. tech., LRI, Université de Paris-Sud, mai 2008.

[20] C. BASTOUL, A. COHEN, S. GIRBAL, S. SHARMA et O. TEMAM : Putting polyhedral loop transformations to work. Dans L. RAUCHWERGER, éd. : *Languages and Compilers for Parallel Computing*, vol. 2958 de *Lecture Notes in Computer Science*, p. 209–225. Springer Berlin Heidelberg, 2004.

[21] BDTI : *FPGAs for DSP*. Berkeley Design Technology, Inc., seconde édition, 2006.

[22] M.-W. BENABDERRAHMANE, L.-N. POUCHET, A. COHEN et C. BASTOUL : The polyhedral model is more widely applicable than you think. *Lecture Notes in Computer Science*, 6011/2010:283–303, 2010.

[23] A. BENVENISTE, P. CASPI, S. A. EDWARDS, N. HALBWACHS, P. LE GUERNIC et R. de SIMONE : The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, janvier 2003.

[24] A. BENVENISTE, P. LE GUERNIC et C. JACQUEMOT : Synchronous programming with events and relations : the Signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[25] G. BERRY : *Pourquoi et comment le monde devient numérique*. Fayard, janvier 2008.

[26] G. BERRY et E. SENTOVICH : Multiclock Esterel. Dans *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, p. 110–125, London, Royaume-Uni, 2001. Springer-Verlag.

[27] B. BHATTACHARYA et S. S. BHATTACHARYYA : *Parameterized Modeling and Scheduling for Dataflow Graphs*. Rapport technique 99-73, Institute for Advanced Computer Studies, University of Maryland, décembre 1999.

[28] B. BHATTACHARYA et S. S. BHATTACHARYYA : Parameterized dataflow modeling of DSP systems. Dans *Proceedings of IEEE International Conference on the Acoustics, Speech, and Signal Processing (ICASSP'00)*, p. 3362–3365, Washington, DC, États-Unis, 2000. IEEE Computer Society.

[29] B. BHATTACHARYA et S. S. BHATTACHARYYA : *Consistency Analysis of Reconfigurable Dataflow Specifications*, vol. 2268 de *Lecture Notes in Computer Science*, chap. 1 dans Embedded Processor Design Challenges, p. 1–17. Springer-Verlag New York, Inc., New York, NY, États-Unis, 2002.

[30] S. S. BHATTACHARYYA, E. A. LEE et P. K. MURTHY : *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, États-Unis, 1996.

[31] G. BILSEN, M. ENGELS, R. LAUWEREINS et J. PEPERSTRAETE : Cyclo-static dataflow. Dans *IEEE Transactions on Signal Processing*, vol. 4, février 1996.

[32] U. BONDHUGULA, A. HARTONO, J. RAMANUJAM et P. SADAYAPPAN : A practical automatic polyhedral parallelizer and locality optimizer. Dans *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'08)*, p. 101–113, New York, NY, États-Unis, 2008. ACM.

[33] J. BOUCARON : *Modélisation formelle de systèmes insensibles à la latence et ordonnancement*. Thèse de doctorat, Université de Nice Sophia Antipolis, décembre 2007.

[34] J. BOUCARON et A. COADOU : *Dynamic Variable Stage Pipeline : an Implementation of its Control*. Rapport de recherche 6918, INRIA, mai 2009.

[35] J. BOUCARON, A. COADOU et R. de SIMONE : *Kpassa : a Tool for Simulating Periodically Scheduled SoCs*. University booth, conférence DATE, avril 2009. http ://www.date-conference.com/date09/files/file/09-ubooth/Session8/S81.pdf.

[36] J. BOUCARON, A. COADOU et R. de SIMONE : *Throughput and FIFO Sizing : an Application to Latency-Insensitive Design*. Rapport de recherche 6919, INRIA, mai 2009.

[37] J. BOUCARON, A. COADOU, B. FERRERO, J.-V. MILLO et R. de SIMONE : *Kahn-Extended Event Graphs*. Rapport de recherche 6541, INRIA, mai 2008.

[38] J. BOUCARON, J.-V. MILLO et R. de SIMONE : Another glance at relay stations in latency-insensitive design. *Electronic Notes in Theoretical Computer Science*, 146(2):41–59, 2006. Proceedings of the Second Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS'05).

[39] J. BOUCARON, J.-V. MILLO et R. de SIMONE : Latency-insensitive design and central repetitive scheduling. Dans *Proceedings of the 4th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'06)*, p. 175–183, Napa Valley, CA, États-Unis, juillet 2006. IEEE Press.

[40] J. BOUCARON, J.-V. MILLO et R. de SIMONE : Formal methods for scheduling of latency-insensitive designs. *EURASIP Journal on Embedded Systems*, 2007(1), 2007.

[41] P. BOULET : *Array-OL Revisited, Multidimensional Intensive Signal Processing Specification*. Rapport de recherche 6113, INRIA, février 2007.

[42] P. BOULET : *Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing*. Rapport de recherche 6467, INRIA, avril 2008.

[43] P. BRISK et M. SARRAFZADEH : Interference graphs for procedures in static single information form are interval graphs. Dans *Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES'07)*, p. 101–110. ACM, 2007.

[44] C. BROOKS, E. A. LEE, X. LIU, S. NEUENDORFFER, Y. ZHAO et H. ZHENG : *Heterogeneous Concurrent Modeling and Design in Java*. Rap. tech. UCB/EECS-2008-28, EECS Department, University of California, Berkeley, avril 2008. vol. 1 : Introduction to Ptolemy II.

[45] J. T. BUCK : *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Thèse de doctorat, University of California, Berkeley, CA, États-Unis, 1993.

[46] J. T. BUCK : Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. Dans *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, vol. 1, p. 508–513, 1994.

[47] J. CARLIER et P. CHRÉTIENNE : *Problème d'Ordonnancement : Modélisation, Complexité, Algorithmes*. Masson, Paris, France, 1988.

[48] L. P. CARLONI : The role of back-pressure in implementing latency-insensitive systems. *Electronic Notes in Theoretical Computer Science*, 146(2):61–80, 2006. Proceedings of the 2nd Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS'05).

[49] L. P. CARLONI, K. L. MCMILLAN, A. SALDANHA et A. L. SANGIOVANNI-VINCENTELLI : A methodology for correct-by-construction latency-insensitive design. Dans *Proceedings of the International Conference on Computer-Aided Design (ICCAD'99)*, p. 309–315, Piscataway, NJ, États-Unis, novembre 1999. IEEE.

[50] L. P. CARLONI, K. L. MCMILLAN et A. L. SANGIOVANNI-VINCENTELLI : Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, septembre 2001.

[51] L. P. CARLONI et A. L. SANGIOVANNI-VINCENTELLI : Performance analysis and optimization of latency insensitive systems. Dans *Proceedings of the 37th Conference on Design Automation (DAC'00)*, p. 361–367, New York, NY, États-Unis, 2000. ACM.

[52] L. P. CARLONI et A. L. SANGIOVANNI-VINCENTELLI : Coping with latency in SOC design. *IEEE Micro*, 22(5):24–35, 2002.

[53] J. CARMONA, J. JULVEZ, J. CORTADELLA et M. KISHINEVSKY : Scheduling synchronous elastic designs. Dans *Proceedings of the 2009 Ninth International Conference on Application of Concurrency to System Design (ACSD'09)*, p. 52–59, Washington, DC, États-Unis, 2009. IEEE Computer Society.

[54] P. CASPI, D. PILAUD, N. HALBWACHS et J. A. PLAICE : Lustre : a declarative language for real-time programming. Dans *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*, p. 178–188, New York, NY, États-Unis, 1987. ACM.

[55] M. R. CASU et L. MACCHIARULO : A detailed implementation of latency insensitive protocols. Dans *Proceedings of the 1st Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS'03)*, p. 94–103, septembre 2003.

[56] M. R. CASU et L. MACCHIARULO : A new approach to latency insensitive design. *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*, p. 576–581, 2004.

[57] M. R. CASU et L. MACCHIARULO : Adaptive latency-insensitive protocols. *IEEE Design & Test of Computers*, 24(5):442–452, sept.-oct. 2007.

[58] M. R. CASU et L. MACCHIARULO : Adaptive latency insensitive protocols and elastic circuits with early evaluation : a comparative analysis. Dans *Proceedings of the 4th Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS'09)*, avril 2009.

[59] N. V. CHERNIKOVA : Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[60] P. CHRÉTIENNE, E. G. J. COFFMAN, J. K. LENSTRA et Z. LIU, éds. *Scheduling Theory and its Applications*. John Wiley & Sons, 1995.

[61] P. CLAUSS : Counting solutions to linear and nonlinear constraints through Ehrhart polynomials : applications to analyze and transform scientific programs. Dans *Proceedings of the 10th International Conference on Supercomputing (ICS'96)*, p. 278–285, New York, NY, États-Unis, 1996. ACM.

[62] A. COHEN, M. DURANTON, C. EISENBEIS, C. PAGETTI, F. PLATEAU et M. POUZET : Synchronization of periodic clocks. Dans *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT'05)*, p. 339–342, septembre 2005.

[63] A. COHEN, M. DURANTON, C. EISENBEIS, C. PAGETTI, F. PLATEAU et M. POUZET : N-synchronous Kahn networks : a relaxed model of synchrony for real-time systems. Dans *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, p. 180–193, New York, NY, États-Unis, janvier 2006. ACM.

[64] A. COHEN, L. MANDEL, F. PLATEAU et M. POUZET : Abstraction of clocks in synchronous data-flow systems. *Lecture Notes in Computer Science*, 5356/2008:237–254, 2008.

[65] A. COHEN, M. SIGLER, S. GIRBAL, O. TEMAM, D. PARELLO et N. VASILACHE : Facilitating the search for compositions of program transformations. Dans *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*, p. 151–160, New York, NY, États-Unis, 2005. ACM.

[66] F. COMMONER, A. W. HOLT, S. EVEN et A. PNUELI : Marked directed graph. *Journal of Computer and System Sciences*, 5:511–523, octobre 1971.

[67] L. COMTET : *Analyse combinatoire*, vol. 1. Presses universitaires de France, 1970.

[68] T. CORMEN, C. LEISERSON, D. RIVEST et C. STEIN : *Introduction à l'algorithmique*. Dunod, seconde édition, 2004. Traduction de l'américain par Xavier Cazin et Georges-Louis Kocher.

[69] J. CORTADELLA et M. KISHINEVSKY : Synchronous elastic circuits with early evaluation and token counterflow. Dans *Proceedings of the 44th Annual Conference on Design Automation (DAC'07)*, p. 416–419, New York, NY, États-Unis, 2007. ACM.

[70] J. CORTADELLA, M. KISHINEVSKY et B. GRUNDMANN : Synthesis of synchronous elastic architectures. Dans *Proceedings of the 43rd Annual Conference on Design Automation (DAC'06)*, p. 657–662, New York, NY, États-Unis, 2006. ACM.

[71] A. DARTE : Understanding loops : the influence of the decomposition of Karp, Miller, and Winograd. Dans *Proceedings of the Eighth ACM/IEEE International Conference of Formal Methods and Models for Codesign (MEMOCODE'2010)*, p. 139–148, Piscataway, NJ, États-Unis, 2010. IEEE Press.

[72] A. DARTE, Y. ROBERT et F. VIVIEN : *Scheduling and Automatic Parallelization*. Birkhaüser Boston, 2000.

[73] A. DARTE, R. S. SCHREIBER et G. VILLARD : Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[74] A. DASDAN et R. K. GUPTA : Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:889–899, 1998.

[75] S. DERRIEN : *Étude quantitative des techniques de partitionnement de réseaux de processeurs pour l'implantation sur circuits FPGA*. Thèse de doctorat, Université de Rennes I, décembre 2002.

[76] A. S. DHODAPKAR et J. E. SMITH : Managing multi-configuration hardware via dynamic working set analysis. *ACM SIGARCH Computer Architecture News*, 30(2):233–244, 2002.

[77] V. DIEKERT et Y. MÉTIVIER : *Partial commutation and traces*, vol. 3, chap. 8 dans *Handbook of Formal Languages : Beyond Words*, p. 457–533. Springer-Verlag New York, Inc., New York, NY, États-Unis, 1997.

[78] P. DUMONT : *Spécification multidimensionnelle pour le traitement du signal systématique*. Thèse de doctorat, Université de Lille I, décembre 2005.

[79] F. Dupont de DINECHIN : *Systèmes structurés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications*. Thèse de doctorat, Université de Rennes I, janvier 1997.

[80] E. EHRHART : Sur les polyèdres rationnels homothétiques à *n* dimensions. *Compte-rendus de l'académie des sciences*, 254:616–618, 1962.

[81] M. ENGELS, G. BILSEN, R. LAUWEREINS et J. A. PEPERSTRAETE : Cyclo-static dataflow : Model and implementation. Dans *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, vol. 1, p. 503–507, Pacific Grove, CA, États-Unis, 1994.

[82] J. FABRI : Automatic storage optimization. Dans *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, p. 83–91. ACM, 1979.

[83] J. FARKAS : Über die theorie der einfachen ungleichungen. *Journal für die Reine und Angewandte Mathematik*, 124:1–27, 1902.

[84] P. FEAUTRIER : Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3): 243–268, 1988.

[85] P. FEAUTRIER : Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, février 1991.

[86] P. FEAUTRIER : Some efficient solutions to the affine scheduling problem : Part I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[87] P. FEAUTRIER : Some efficient solutions to the affine scheduling problem : Part II, multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[88] P. FEAUTRIER : Scalable and modular scheduling. Dans A. PIMENTEL et S. VASSILIADIS, éds : *Computer Systems : Architectures, Modeling, and Simulation*, vol. 3133 de *Lecture Notes in Computer Science*, p. 233–254. Springer Berlin Heidelberg, 2004.

[89] C. J. FIDGE : Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, février 1988.

[90] M. GALCERAN-OMS, J. CORTADELLA et M. KISHINEVSKY : Speculation in elastic systems. Dans *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*, p. 292–295, New York, NY, États-Unis, juillet 2009. ACM.

[91] D. GALE, H. W. KUHN et A. W. TUCKER : *Linear Programming and the Theory of Games*, chap. XII dans Koopmans (éd.) : Activity Analysis of Production and Allocation, p. 317–329. Wiley, première édition, 1951.

[92] G. R. GAO, R. GOVINDARAJAN et P. PANANGADEN : Well-behaved dataflow programs for DSP computation. *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'92)*, 5:561–564, mars 1992.

[93] GAUTAM, D. KIM et S. RAJOPADHYE : Scheduling in the Z-polyhedral model. Dans *IEEE International Parallel and Distributed Processing Symposium 2007 (IPDPS'07)*, 2007.

[94] GAUTAM et S. RAJOPADHYE : The Z-polyhedral model. Dans *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, p. 237–248, New York, NY, États-Unis, 2007. ACM.

[95] M. GEILEN et T. BASTEN : Requirements on the execution of Kahn process networks. Dans *Proceedings of the 12th European conference on Programming (ESOP'03)*, p. 319–334, Berlin, Heidelberg, 2003. Springer-Verlag.

[96] M. GEILEN et T. BASTEN : Reactive process networks. Dans *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT'04)*, p. 137–146, New York, NY, États-Unis, 2004. ACM.

[97] H. J. GENRICH : *Einfache Nicht-Sequentielle Prozesse*. Thèse de doctorat, Rheinische Friedrich-Wilhelms-Universität, Bonn, Allemagne, 1970.

[98] A. GIRAULT, B. LEE et E. A. LEE : Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, juin 1999.

[99] S. GIRBAL, N. VASILACHE, C. BASTOUL, A. COHEN, D. PARELLO, M. SIGLER et O. TEMAM : Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[100] C. GLITIA : *Optimisation des applications de traitement systématique intensives sur system-on-chip*. Thèse de doctorat, Université de Lille I, novembre 2009.

[101] M. C. GOLUMBIC : *Algorithmic Graph Theory and Perfect Graphs*, vol. 57 de *Annals of Discrete Mathematics*. North-Holland Publishing Co., Amsterdam, Pays-Bas, 2004.

[102] M. I. GORDON : *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. Thèse de doctorat, MIT, Cambridge, MA, États-Unis, juin 2010.

[103] R. GOVINDARAJAN et G. R. GAO : Rate-optimal schedule for multi-rate DSP computations. *Journal of VLSI Signal Processing Systems*, 9(3):211–232, 1995.

[104] A. GREENFIELD : *Everyware : The Dawning Age of Ubiquitous Computing*. New Riders Publishing, première édition, mars 2006.

[105] M. GRIEBL : *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, juin 2004. Thèse d'habilitation.

[106] M. GRIEBL, P. FABER et C. LENGAUER : Space–time mapping and tiling : a helpful combination. *Concurrency and Computation : Practice & Experience*, 16(2-3):221–246, 2004.

[107] M. GRIEBL, P. FEAUTRIER et C. LENGAUER : Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[108] M. GRIEBL et C. LENGAUER : The loop parallelizer LooPo. Dans *Proceedings of the 6th Workshop on Compilers for Parallel Computers (CPC'96)*, num. 21, p. 311–320. Forschungszentrum Jülich, 1996.

[109] A. GRÖSSLINGER : Precise management of scratchpad memories for localising array accesses in scientific codes. Dans *Proceedings of the 18th International Conference on Compiler Construction (CC'09)*, p. 236–250, Berlin, Heidelberg, 2009. Springer-Verlag.

[110] P. GRUN, P. ELES, K. KUCHCINSKI et Z. PEN : Automatic parallelization of a petri net-based design representation for high-level synthesis. Dans *Proceedings of the 22nd EUROMICRO Conference*, p. 185–192, Los Alamitos, CA, États-Unis, septembre 1996. IEEE Computer Society.

[111] P. GRUN, A. NICOLAU et N. DUTT : *Memory Architecture Exploration for Programmable Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, États-Unis, 2002.

[112] D. GRUND et S. HACK : A fast cutting-plane algorithm for optimal coalescing. Dans S. KRISHNAMURTHI et M. ODERSKY, éds : *Compiler Construction 2007*, vol. 4420 de *Lecture Notes In Computer Science*, p. 111 – 125. Springer, 2007.

[113] L. J. GUIBAS et R. SEDGEWICK : A dichromatic framework for balanced trees. Dans *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS'78)*, p. 8–21, Washington, DC, États-Unis, 1978. IEEE Computer Society.

[114] R. GUPTA et F. BREWER : *High-Level Synthesis : A Retrospective*, chap. 2 dans *High-Level Synthesis*, p. 13–28. Springer Netherlands, 2008.

[115] F. HAIM, M. SEN, D.-I. KO, S. S. BHATTACHARYYA et W. WOLF : Mapping multimedia applications onto configurable hardware with parameterized cyclo-static dataflow graphs. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'2006)*, 3:1052–1055, mai 2006.

[116] Z. HANZÀLEK : Algorithm modelling with Petri nets – comparison with data dependence graphs. Dans *IEEE International Conference on Systems, Man, and Cybernetics (SMC'98)*, vol. 1, p. 214–219, Piscataway, NJ, États-Unis, octobre 1998. IEEE Press.

[117] D. HARRIS : *Local Stall Propagation*. Harvey Mudd College, septembre 2000.

[118] P. C. HELD : *Functional Design of Data-Flow Networks*. Thèse de doctorat, Technische Universiteit Delft, Delft, Pays-Bas, mai 1996.

[119] INTEL CORPORATION : *New Intel(R) Xeon(R) Processor Pushes Mission Critical into the Mainstream*. Communiqué de presse, mars 2010. http ://www.intel.com/pressroom/archive/releases/2010/20100330comp_sm.htm.

[120] IRISA et UNIVERSITÉ DE STRASBOURG : *Polylib – a library of polyhedral functions*. http ://www.irisa.fr/polylib/.

[121] ITRS : *The International Technology Roadmap for Semiconductors, 2005 Edition*, 2005.

[122] ITRS : *The International Technology Roadmap for Semiconductors, 2007 Edition*, 2007.

[123] D. B. JOHNSON : Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[124] D. B. JOHNSON : Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[125] B. JOINNAULT : *Conception d'algorithmes et d'architectures systoliques*. Thèse de doctorat, Université de Rennes I, septembre 1987.

[126] G. KAHN : The semantics of a simple language for parallel programming. Dans J. L. ROSENFELD, éd. : *Information Processing '74 : Proceedings of the IFIP Congress*, p. 471–475, New York, NY, États-Unis, 1974. North-Holland.

[127] R. M. KARP et R. E. MILLER : Properties of a model for parallel computations : Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[128] R. M. KARP, R. E. MILLER et S. WINOGRAD : The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[129] A. KERIHUEL, R. MCCONNELL et S. RAJOPADHYE : *VSDF : Synchronous Data Flow for VLSI*. Rapport de recherche 2237, INRIA, juin 1994.

[130] C. W. KESSLER : Parallel Fourier-Motzkin elimination. Dans *Proceedings of the Second International Euro-Par Conference on Parallel Processing (Euro-Par'96)*, vol. 2, p. 66–71, London, Royaume-Uni, 1996. Springer-Verlag.

[131] A. C. J. KIENHUIS : *Design Space Exploration of Stream-based Dataflow Architectures : Methods and Tools*. Thèse de doctorat, Delft University of Technology, Pays-Bas, janvier 1999.

[132] B. KIENHUIS : *MatParser : An Array Dataflow Analysis Compiler*. Rap. tech. UCB/ERL M00/9, University of California, Berkeley, Berkeley, CA, États-Unis, février 2000.

[133] B. KIENHUIS, E. RIJPKEMA et E. DEPRETTERE : Compaan : deriving process networks from matlab for embedded signal processing architectures. Dans *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES'00)*, p. 13–17, New York, NY, États-Unis, 2000. ACM.

[134] D. E. KNUTH : *Foreword*, chap. dans $A = B$, p. vii. A. K. Peters, Ltd., janvier 1996.

[135] D. E. KNUTH : *The Art of Computer Programming*, vol. 3 : Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, États-Unis, seconde édition, 1998.

[136] D.-I. KO : *System Synthesis for Image Processing Applications*. Thèse de doctorat, University of Maryland, 2006.

[137] D.-I. KO et S. S. BHATTACHARYYA : Dynamic configuration of dataflow graph topology for DSP system design. *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'05)*, 5:69–72, mars 2005.

[138] D. L. KUCK : *Structure of Computers and Computations*. John Wiley & Sons, Inc., New York, NY, États-Unis, 1978.

[139] H. W. KUHN et A. W. TUCKER : Nonlinear programming. Dans J. NEYMAN, éd. : *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, p. 481–492, Berkeley, CA, États-Unis, 1951. University of California Press.

[140] I. KUON et J. ROSE : Measuring the gap between FPGAs and ASICs. Dans *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA'06)*, p. 21–30, New York, NY, États-Unis, 2006. ACM.

[141] L. LAMPORT : Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[142] H. LE VERGE : *A Note on Cherniakova's Algorithm*. Rapport de recherche 1662, INRIA, avril 1992.

[143] E. A. LEE et D. G. MESSERSCHMITT : Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, 1987.

[144] E. A. LEE et D. G. MESSERSCHMITT : Synchronous data flow. *Proceeding of the IEEE*, 75(9):1235–1245, 1987.

[145] E. A. LEE et T. M. PARKS : Dataflow process networks. p. 59–85, 2002.

[146] D. H. LEHMER : Teaching combinatorial tricks to a computer. *Proceedings of Symposia in Applied Mathematics*, vol. 10 : Combinatorial Analysis:179–193, 1960.

[147] C. E. LEISERSON et J. B. SAXE : Optimizing synchronous systems. *Annual IEEE Symposium on Foundations of Computer Science*, p. 23–36, 1981.

[148] C. E. LEISERSON et J. B. SAXE : Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[149] C. LENGAUER : Loop parallelization in the polytope model. Dans *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, p. 398–416, Londres, Royaume-Uni, 1993. Springer-Verlag.

[150] C.-H. LI et L. P. CARLONI : Using functional independence conditions to optimize the performance of latency-insensitive systems. Dans *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'07)*, p. 32–39, Piscataway, NJ, États-Unis, 2007. IEEE Press.

[151] C.-H. LI, R. L. COLLINS, S. SONALKAR et L. P. CARLONI : Design, implementation, and validation of a new class of interface circuits for latency-insensitive design. Dans *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'07)*, p. 13–22, Washington, DC, États-Unis, mai 2007. IEEE Computer Society.

[152] A. W. LIM : *Improving Parallelism And Data Locality With Affine Partitioning*. Thèse de doctorat, Stanford University, Computer Science Department, septembre 2001.

[153] V. LOECHNER, B. MEISTER et P. CLAUSS : Precise data locality optimization of nested loops. *The Journal of Supercomputing*, 21(1):37–76, 2002.

[154] V. LOECHNER et D. K. WILDE : Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, 1997.

[155] A. LORANGER : *Dictionnaire biographique et historique de la micro-informatique*. Éditions MultiMondes, 2000.

[156] R. LU et C.-K. KOH : Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. Dans *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'03)*, p. 227–231, Washington, DC, États-Unis, 2003. IEEE Computer Society.

[157] L. MANDEL et F. PLATEAU : Abstraction d'horloges dans les systèmes synchrones flot de données. Dans *Vingtièmes Journées Francophones des Langages Applicatifs (JFLA'09)*, Saint-Quentin sur Isère, France, février 2009.

[158] L. MANDEL, F. PLATEAU et M. POUZET : Lucy-n : une extension n-synchrone de Lustre. Dans *Vingt et unièmes Journées Francophones des Langages Applicatifs (JFLA'10)*, janvier 2010.

[159] O. MARCHETTI et A. MUNIER-KORDON : Minimizing place capacities of weighted event graphs for enforcing liveness. *Discrete Event Dynamic Systems*, 18(1):91–109, 2008.

[160] O. MARCHETTI et A. MUNIER-KORDON : A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research*, 197(2):532–540, septembre 2009.

[161] A. B. MATOS : Periodic sets of integers. *Theoretical Computer Science*, 127(2):287–312, 1994.

[162] F. MATTERN : Virtual time and global states of distributed systems. Dans *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, p. 215–226. North-Holland, octobre 1988.

[163] C. MAURAS : *Définition de Alpha : un langage pour la programmation systolique*. Rapport de recherche 1090, INRIA, 1989.

[164] C. MAURAS, P. QUINTON, S. RAJOPADHYE et Y. SAOUTER : Scheduling affine parameterized recurrences by means of variable dependent timing functions. Dans *Proceedings of the International Conference on Application Specific Array Processors*, p. 100–110, septembre 1990.

[165] MÉLANGE GROUP, CSU : AlphaZ homepage. http ://www.cs.colostate.edu/AlphaZ/.

[166] J.-V. MILLO : *Ordonnancements périodiques dans les réseaux de processus : application à la conception insensible aux latences*. Thèse de doctorat, Université de Nice Sophia Antipolis, décembre 2008.

[167] R. MILNER : *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, États-Unis, 1982.

[168] R. MILNER : *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, États-Unis, 1989.

[169] H. MINKOWSKI : *Geometrie der Zahlen*. Teubner, Leipzig, Allemagne, première édition, 1896.

[170] H. MINKOWSKI : Raum und zeit. Dans *Jahresberichte der Deutschen Mathematiker-Vereinigung*, Leipzig, Allemagne, 1909. BG Teubner.

[171] K. MORIN-ALLORY : *Vérification formelle dans le modèle polyédrique*. Thèse de doctorat, Université de Rennes I, octobre 2004.

[172] T. S. MOTZKIN, H. RAIFFA, G. L. THOMPSON et R. M. THRALL : *The Double Description Method*, vol. 2 de *Annals of Mathematics Studies*, chap. dans Contributions to the Theory of Games, p. 51–73. Princeton University Press, Princeton, NY, États-Unis, mars 1953.

[173] S. S. MUCHNICK : *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, États-Unis, 1997.

[174] T. MURATA : Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, avril 1989.

[175] P. K. MURTHY : *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. Thèse de doctorat, University of California, Berkeley, Berkeley, CA, États-Unis, 1996.

[176] P. K. MURTHY et E. A. LEE : *A Generalization of Multidimensional Synchronous Dataflow to Arbitrary Sampling Lattices*. Rap. tech. UCB/ERL M95/59, University of California, Berkeley, mars 1995.

[177] G. L. NABER : *The Geometry of Minkowski Spacetime : An Introduction to the Mathematics of the Special Theory of Relativity*. Courier Dover Publications, 2003.

[178] V. K. NANDIVA : *Advances in Register Allocation Techniques*, chap. 21 dans The Compiler Design Handbook : Optimizations and Machine Code Generation. CRC Press, seconde édition, 2007.

[179] M. NEBUT : *Réactions synchrones : spécification et analyse*. Thèse de doctorat, Université de Rennes I, novembre 2002.

[180] S. D. NIKOLOPOULOS : Coloring permutation graphs in parallel. *Discrete Applied Mathematics*, 120(1-3):165–195, 2002.

[181] H. NIKOLOV, T. STEFANOV et E. DEPRETTERE : Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, mars 2008.

[182] M. PALKOVIC, F. CATTHOOR et H. CORPORAAL : Trade-offs in loop transformations. *ACM Transactions on Design Automation of Electronic Systems*, 14(2):1–30, 2009.

[183] C. PARK, J. CHUNG et S. HA : Efficient dataflow representation of MPEG-1 audio (layer iii) decoder algorithm with controlled global states. *IEEE Workshop on Signal Processing Systems (SiPS'99)*, p. 341–350, octobre 1999.

[184] C. PARK, J. CHUNG et S. HA : Extended synchronous dataflow for efficient DSP system prototyping. Dans *Proceedings of the Tenth IEEE International Workshop on Rapid System Prototyping (RSP'99)*, p. 196, Washington, DC, États-Unis, juin 1999. IEEE Computer Society.

[185] C. PARK et S. HA : Hardware synthesis from SPDF representation for multimedia applications. Dans *Proceedings of the 13rd International Symposium on System Synthesis (ISSS'00)*, p. 215–220, Washington, DC, États-Unis, 2000. IEEE Computer Society.

[186] T. M. PARKS : *Bounded Scheduling of Process Networks*. Thèse de doctorat, University of California, Berkeley, Berkeley, CA, États-Unis, décembre 1995.

[187] C. A. PETRI : *Kommunikation mit Automaten*. Thèse de doctorat, Institut für instrumentelle Mathematik, Bonn, Allemagne, 1962.

[188] E. PIEL : *Ordonnancement de systèmes parallèles temps-réel – De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles*. Thèse de doctorat, Université de Lille I, décembre 2007.

[189] J. L. PINO, E. A. LEE et S. S. BHATTACHARYYA : A hierarchical multiprocessor scheduling system for DSP applications. Dans *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (ASILOMAR'95)*, vol. 2, p. 122, Washington, DC, États-Unis, 1995. IEEE Computer Society.

[190] F. PLATEAU : *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. Thèse de doctorat, Université de Paris XI, janvier 2010.

[191] W. PLISHKER, N. SANE et S. S. BHATTACHARYYA : A generalized scheduling approach for dynamic dataflow applications. Dans *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*, Washington, DC, États-Unis, 2009. IEEE Computer Society.

[192] D. POTOP-BUTUCARU, R. de SIMONE et J.-P. TALPIN : *The Synchronous Hypothesis and Synchronous Languages*, chap. 8 dans *Embedded Systems Handbook*. CRC Press, 2006.

[193] D. POTOP-BUTUCARU, S. A. EDWARDS et G. BERRY : *Compiling Esterel*. Springer, 2007.

[194] L.-N. POUCHET : *Iterative Optimization in the Polyhedral Model*. Thèse de doctorat, Université de Paris-Sud XI, Paris, janvier 2010.

[195] L.-N. POUCHET, C. BASTOUL, A. COHEN et J. CAVAZOS : Iterative optimization in the polyhedral model : Part ii, multidimensional time. Dans *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, p. 90–100, New York, NY, États-Unis, 2008. ACM.

[196] L.-N. POUCHET, C. BASTOUL, A. COHEN et N. VASILACHE : Iterative optimization in the polyhedral model : Part i, one-dimensional time. Dans *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*, p. 144–156, Washington, DC, États-Unis, 2007. IEEE Computer Society.

[197] W. PUGH et D. WONNACOTT : Eliminating false data dependences using the omega test. Dans *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI'92)*, p. 140–151, New York, NY, États-Unis, 1992. ACM.

[198] P. QUINTON, S. RAJOPADHYE et T. RISSET : *On Manipulating Z-Polyhedra*. Publication interne 1016, IRISA, juillet 1996.

[199] I. RADOJEVIC, Z. SALCIC et P. ROOP : Design of heterogeneous embedded systems using DFCharts model of computation. Dans *Proceedings of the 19th International Conference on VLSI Design (VLSID'06)*, p. 461–464, Washington, DC, États-Unis, 2006. IEEE Computer Society.

[200] C. RAMCHANDANI : *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Thèse de doctorat, MIT, Cambridge, MA, États-Unis, septembre 1973.

[201] R. REITER : Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, 1968.

[202] M. D. RIEDEL : *Cyclic Combinational Circuits*. Thèse de doctorat, California Institute of Technology, Pasadena, CA, États-Unis, mai 2004.

[203] S. RITZ, M. PANKERT, V. ŽIVOJINOVIĆ et H. MEYR : High-level software synthesis for the design of communication systems. *IEEE Journal on Selected Areas in Communications*, 11(3):348–358, avril 1993.

[204] S. RITZ, M. PANKERT, V. ŽIVOJINOVIĆ et H. MEYR : Optimum vectorization of scalable synchronous dataflow graphs. *International Conference on Application-Specific Array Processors*, p. 285–296, octobre 1993.

[205] S. M. ROBINSON : A quadratically-convergent algorithm for general nonlinear programming problems. *Mathematical Programming*, 3(1):145–156, 1972.

[206] T. SASAKI, Y. ICHIKAWA, T. HIRONAKA, T. KITAMURA et T. KONDO : Evaluation of low-energy and high-performance processor using variable stages pipeline technique. *IET Computers & Digital Techniques*, 2(3):230–238, mai 2008.

[207] R. S. SCHREIBER, B. R. RAU, S. ADITYA GUPTA, V. K. KATHAIL et S. ANIK : Programmatic synthesis of processor element arrays. US Patent 6507947B1, janvier 2003.

[208] R. S. SCHREIBER, B. R. RAU et A. DARTE : Programmatic iteration scheduling for parallel processors. US Patent 6438747B1, août 2002.

[209] A. SCHRIJVER : *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, États-Unis, 1986.

[210] R. SEGHIR : *Méthodes de dénombrement de points entiers de polyèdres et applications à l'optimisation de programmes*. Thèse de doctorat, Université de Strasbourg I, décembre 2006.

[211] M. SEN, I. CORRETJER, F. HAIM, S. SAHA, S. S. BHATTACHARYYA, J. SCHLESSMAN et W. WOLF : Computer vision on FPGAs : Design methodology and its application to gesture recognition. Dans *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, p. 133, Washington, DC, États-Unis, 2005. IEEE Computer Society.

[212] M. SEN, I. CORRETJER, F. HAIM, S. SAHA, J. SCHLESSMAN, T. LV, S. S. BHATTACHARYYA et W. WOLF : Dataflow-based mapping of computer vision algorithms onto FPGAs. *EURASIP Journal on Embedded Systems*, 2007(1):29–29, 2007.

[213] H. SHIMADA, H. ANDO et T. SHIMADA : Pipeline stage unification : a low-energy consumption technique for future mobile processors. Dans *Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISLPED'03)*, p. 326–329, New York, NY, États-Unis, 2003. ACM.

[214] G. C. SIH : *Multiprocessor Scheduling to Account for Interprocessor Communication*. Thèse de doctorat, University of California, Berkeley, Berkeley, CA, États-Unis, 1991.

[215] M. SINGH et M. THEOBALD : Generalized latency-insensitive systems for single-clock and multi-clock architectures. Dans *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*, p. 1008–1013, Washington, DC, États-Unis, 2004. IEEE Computer Society.

[216] I. M. SMARANDACHE : *Transformations affines d'horloges : application au codesign de systèmes temps-réel en utilisant les langages Signal et Alpha*. Thèse de doctorat, Université de Rennes I, octobre 1997.

[217] C. SOVIANI et S. A. EDWARDS : FIFO sizing for high-performance pipelines. Dans *Proceedings of the 16th International Workshop on Logic and Synthesis (IWLS'07)*, juin 2007.

[218] T. STEFANOV : *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. Thèse de doctorat, Universiteit Leiden, Leyde, Pays-Bas, décembre 2004.

[219] T. STEFANOV, C. ZISSULESCU, A. TURJAN, B. KIENHUIS et E. DEPRETTERE : System design using Kahn process networks : The Compaan/Laura approach. Dans *Proceedings of the conference on Design, automation and test in Europe (DATE'04)*, Washington, DC, États-Unis, 2004. IEEE Computer Society.

[220] J. E. STINE, I. CASTELLANOS, M. WOOD, J. HENSON, F. LOVE, W. R. DAVIS, P. D. FRANZON, M. BUCHER, S. BASAVARAJAIAH, J. OH et R. JENKAL : FreePDK : An open-source variation-aware design kit. Dans *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, p. 173–174, Washington, DC, États-Unis, 2007. IEEE Computer Society.

[221] J. E. STINE, J. GRAD, I. CASTELLANOS, J. BLANK, V. DAVE, M. PRAKASH, N. ILIEV et N. JACHIMIEC : A framework for high-level synthesis of system-on-chip designs. Dans *Proceedings of the 2005 IEEE International Conference on Microelectronic Systems Education (MSE'05)*, p. 67–68, Washington, DC, États-Unis, 2005. IEEE Computer Society.

[222] K. J. SUPOWIT : Decomposing a set of points into chains, with applications to permutation and circle graphs. *Information Processing Letters*, 21(5):249–252, 1985.

[223] SYNFORA, INC. : *Writing C Applications : Developper's Guide*, 2007. Documentation de Pico Express.

[224] W. THIES, J. LIN et S. AMARASINGHE : *Phased Computation Graphs in the Polyhedral Model*. Rap. tech. LCS-TM-630, MIT Laboratory for Computer Science, Cambridge, MA, États-Unis, août 2002.

[225] W. THIES, F. VIVIEN et S. AMARASINGHE : A step towards unifying schedule and storage optimization. *ACM Transactions on Programming Languages and Systems*, 29(6):34 :1–45, 2007.

[226] A. TURJAN : *Compiling Nested Loop Programs to Process Networks*. Thèse de doctorat, Universiteit Leiden, Leyde, Pays-Bas, mars 2007.

[227] A. TURJAN, B. KIENHUIS et E. DEPRETTERE : *Realizations Of The Extended Linearization Model*, chap. 9 dans *Domain-Specific Processors : Systems, Architectures, Modeling, and Simulation*, p. 171–191. Marcel Dekker, Inc., New York, NY, États-Unis, première édition, 2004.

[228] J. F. A. K. van BENTHEM : *The Logic of Time : A Model-Theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse*, vol. 156 de *Synthese Library*. Springer, seconde édition, 1991.

[229] S. VERDOOLAEGE : *Incremental Loop Transformations and Enumeration of Parametric Sets*. Thèse de doctorat, Katholieke Universiteit Leuven, Louvain, Belgique, avril 2005.

[230] S. VERDOOLAEGE, H. NIKOLOV et T. STEFANOV : pn : a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(1), 2007.

[231] P. WAUTERS, M. ENGELS, R. LAUWEREINS et J. A. PEPERSTRAETE : Cyclo-dynamic dataflow. Dans *Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP'96)*, p. 319. IEEE Computer Society, 1996.

[232] D. K. WILDE : *A Library for Doing Polyhedral Operations*. Rapport de recherche 2157, INRIA, décembre 1993.

[233] D. K. WILDE : *The Alpha language*. Publication interne 827, IRISA, mai 1994.

[234] J. XUE : Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, mars 1996.

[235] J. YAO, S. MIWA, H. SHIMADA et S. TOMITA : Optimal pipeline depth with pipeline stage unification adoption. *SIGARCH Computer Architecture News*, 35(5):3–9, 2007.

[236] J. YAO, H. SHIMADA, Y. NAKASHIMA, S.-I. MORI et S. TOMITA : Program phase detection based dynamic control mechanisms for pipeline stage unification adoption. Dans *Proceedings of the 6th International Symposium on High-Performance Computing and 1st International Conference on Advanced Low Power Systems (ISHPC'05/ALPS'06)*, p. 494–507, Berlin, Heidelberg, 2008. Springer-Verlag.