Numéro d'ordre : 40135

Université des Sciences et Technologies de Lille

THÈSE

présentée pour obtenir le titre de docteur
spécialité Informatique

par

CALIN GLITIA

OPTIMISATION DES APPLICATIONS DE TRAITEMENT SYSTÉMATIQUE INTENSIVES SUR SYSTEMS-ON-CHIP

Thèse soutenue le 23 novembre 2009, devant la commission d'examen formée de :

Nouredine Melab	Professeur Université de Lille 1	Président
François Irigoin	Maître de Recherche École des Mines Paris	Rapporteur
Patrice Quinton	Professeur ENS Cachan	Rapporteur
Michel Barreteau	Chef de Projet R&D THALES Palaiseau	Examinateur
Sven-Bodo Scholz	Senior Lecturer University of Hertfordshire UK	Examinateur
Pierre Boulet	Professeur Université de Lille 1	Directeur

Université des Sciences et Technologies de Lille
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

OPTIMISATION DES APPLICATIONS DE TRAITEMENT SYSTÉMATIQUE INTENSIVES SUR SYSTEMS-ON-CHIP

CALIN GLITIA

Thèse de doctorat

23 Novembre 2009

ABSTRACT

Intensive signal processing applications appear in many application domains such as video processing or detection systems. These applications handle multidimensional data structures (mainly arrays) to deal with the various dimensions of the data (space, time, frequency). A specification language allowing the direct manipulation of these different dimensions with a high level of abstraction is a key to handling the complexity of these applications and to benefit from their massive potential parallelism. The Array-OL specification language is designed to do just that.

In this thesis, we introduce an extension of Array-OL to express cycle dependences by the way of uniform inter-repetition dependences. We show that this specification language is able to express the main patterns of computation of the intensive signal processing domain. We discuss also the repetitive modeling of parallel applications, repetitive architectures and uniform mappings of the former to the latter, using the Array-OL concepts integrated into the Modeling and Analysis of Real-time and Embedded systems (MARTE) UML profile.

High-level data-parallel transformations are available to adapt the application to the execution, allowing to choose the granularity of the flows and a simple expression of the mapping by tagging each repetition by its execution mode : data-parallel or sequential. The whole set of transformations was reviewed, extended and implemented as a part of the Gaspard2 co-design environment for embedded systems.

With the introduction of the uniform dependences into the specification, our interest turns also on the interaction between these dependences and the high-level transformations. This is essential in order to enable the usage of the refactoring tools on the models with uniform dependences.

Based on the high-level refactoring tools, strategies and heuristics can be designed to help explore the design space. We propose a strategy that allows to find good trade-offs in the usage of storage and computation resources, and in the parallelism (both task and data parallelism) exploitation, strategy illustrated on an industrial radar application.

RÉSUMÉ

Les applications de traitement intensif de signal apparaissent dans de nombreux domaines d'applications tels que multimédia ou systèmes de détection. Ces applications gèrent les structures de données multidimensionnelles (principalement des tableaux) pour traiter les différentes dimensions des données (espace, temps, fréquence). Un langage de spécification permettant l'utilisation directe de ces différentes dimensions avec un haut niveau d'abstraction est une des clés de la manipulation de la complexité de ces applications et permet de bénéficier de leur parallélisme potentiel. Le langage de spécification Array-OL est conçu pour faire exactement cela.

Dans cette thèse, nous introduisons une extension d'Array-OL pour exprimer des dépendances cycliques par des dépendances interrépétitions uniformes. Nous montrons que ce langage de spécification est capable d'exprimer les principaux motifs de calcul du domaine de traitement de signal intensif. Nous discutons aussi de la modélisation répétitive des applications parallèles, des architectures répétitives et les placements uniformes des premières sur les secondes, en utilisant les concepts Array-OL intégrés dans le profil UML MARTE (Modeling and Analysis of Real-time and Embedded systems).

Des transformations de haut niveau data-parallèles sont disponibles pour adapter l'application à l'exécution, ce qui permet de choisir la granularité des flots et une simple expression du placement en étiquetant chaque répétition par son mode d'exécution : data parallèle ou séquentiel. L'ensemble des transformations a été revu, étendu et implémenté dans le cadre de l'environnement de comodélisation pour les systèmes embarqués, Gaspard2.

Avec l'introduction des dépendances uniformes, notre intérêt s'est tourné aussi sur l'interaction entre ces dépendances et les transformations de haut niveau. C'est essentiel, afin de permettre l'utilisation des outils de refactoring sur les modèles avec dépendances uniformes.

En utilisant les outils de refactoring de haut niveau, des stratégies et des heuristiques peuvent être conçues pour aider à l'exploration de l'espace de conception. Nous proposons une stratégie qui permet de trouver de bons compromis entre l'usage de stockage et de ressources de calcul, et dans l'exploitation de parallélisme (à la fois de tâches et de données), stratégie illustrée sur une application industrielle radar.

PUBLICATIONS

Une partie du matériel présenté dans ce chapitre a également été publié dans les publications suivantes :

JOURNAUX [1]

[1] Calin GLITIA, Philippe DUMONT et Pierre BOULET : Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 2009. URL http://springerlink.com/content/w3821760381l4432/?p=fc0a4428f2f4468a9d630d2a434a6f69&pi=0.

CONFÉRENCES INTERNATIONALES À COMITÉ DE LECTURE ET ACTES [2]

[2] Calin GLITIA et Pierre BOULET : High level loop transformations for multidimensional signal processing embedded applications. *In International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS VIII)*, Samos, Greece, juillet 2008. URL http://samos.et.tudelft.nl/samos_viii/.

[3] Calin GLITIA et Pierre BOULET : Interaction between inter-repetition dependences and high-level transformations in array-ol. *In Conference on Design and Architectures for Signal and Image Processing (DASIP 2009)*, Sophia Antipolis, France, septembre 2009.

WORKSHOPS INTERNATIONAUX AUX COMITÉ DE LECTURE [1]

[4] Calin GLITIA : Interaction between delays and high-level transformations in array-ol. *In 2nd Artist Workshop on Models of Computation and Communication*, Eindhoven, Netherlands, juillet 2008.

RAPPORTS DE RECHERCHE [2]

[5] Calin GLITIA et Pierre BOULET : High level loop transformations for systematic signal processing embedded applications. Research Report 6469, INRIA, 03 2008. URL https://hal.inria.fr/inria-00262023.

[6] Calin GLITIA, Pierre BOULET, Éric LENORMAND et Michel BARRETEAU : Repetitive model refactoring strategy for the design space exploration of intensive signal processing applications. Rapport technique number not yet available, INRIA, septembre 2009. extended version.

REMERCIEMENTS

Je tiens à exprimer ma profonde gratitude envers Pierre Boulet pour m'avoir encadré, conseillé et soutenu tout au long de ma thèse et pour m'avoir guidé et aidé lors de l'écriture de ce manuscrit. Je remercie également les rapporteurs de cette thèse, François Irigoin et Patrice Quinton, pour avoir lu et commenté ce manuscrit et aussi pour leur remarques, questions et recommandations.

Je tiens à remercier Nouredine Melab d'avoir accepté de présider mon jury de thèse et je remercie également Michel Barreteau et Sven-Bodo Scholz d'avoir fait partie de mon jury.

Les travaux présentés font tous partie du projet Gaspard2 et ils n'auraient pas pu avoir lieu sans le travail fournit par tous les autres contributeurs de ce projet. J'en profite pour exprimer toute ma sympathie et gratitude à l'ensemble des membres de l'équipe DaRT de l'INRIA Lille-Nord Europe et le Laboratoire d'Informatique Fondamentale de Lille, pour leur aide, leur collaboration, leur solidarité et leur amitié.

Enfin, je souhaiterais remercier ma famille et mes amis pour m'avoir soutenu et aidé au cours de ma thèse.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION

CONTEXTE

Notre monde est massivement parallèle. Depuis un moment le monde de l'informatique s'intéresse de nouveau au parallélisme, car celui-ci est la clef de la performance.

Les applications de l'informatique embarquée se sont étendues et surtout suivent une tendance exigeant de plus en plus de puissance de calcul. La plupart de ces applications correspondent à du traitement de signal : une suite de calculs appliqués sur des flots de données généralement multidimensionnels. Ces traitements sont en essence parallèles et répétitifs. En plus d'être gourmandes en puissance de calcul, ces applications sont souvent soumises à des contraintes de temps réel qu'il convient de respecter.

Dans la conception de systèmes électroniques, la loi de Moore [65] s'applique toujours, tous les trois ans il est possible de placer deux fois plus de transistors sur une puce électronique. Cette capacité à placer de plus en plus de composants sur une puce favorise l'évolution continue des systèmes sur puce (SoC, pour System-on-Chip). De par leur forte capacité d'intégration, les SoC offrent de grandes économies en consommation d'énergie et en espace, ainsi qu'un gain important en performance. La puce n'est plus dédiée à un type de composant particulier, mais contient tous les composants qui forment un système informatique (processeur, mémoire, réseau d'interconnexion, circuits analogiques, etc.). Pour une utilisation efficace du grand nombre de transistors disponible pour le calcul, de multiples processeurs sont intégrés sur la même puce (MPSoC, multiprocesseur SoC) et l'usage du parallélisme de plus en plus massif est essentiel.

Pour exploiter la puissance de calcul des architectures à unités parallèles, le code exécuté a un rôle essentiel : un maximum du parallélisme de la spécification doit se retrouver à l'exécution. Malheureusement, un décalage peut être observé entre les évolutions technologiques et logicielles, marqué par une dégradation de la qualité de la programmation. Comme raison à cela, on peut penser aux complexités architecturales qui font que les compilateurs ne peuvent plus suivre en fournissant du code optimisé au maximum. Pour avoir de bons résultats, le code d'entrée devrait être conçu dans la vision parallèle, facilitant la tâche du compilateur.

Dans le contexte de la conception de SoC, la complexité de conception apporte des contraintes supplémentaires concernant le risque que le produit final ne corresponde pas à la spécification et le délai de production qui les rend parfois obsolètes avant même leur mise sur le marché. À ces problèmes, il convient d'ajouter, une fois de plus, la demande toujours croissante en puissance de calcul.

Parallèlement aux évolutions technologiques, des méthodologies de conception de SoC ont évolué et sont le lien entre le SoC et son concepteur. Un exemple de méthodologie utilisée depuis quelques années déjà est la décomposition en sous-composants interconnectés et la réutilisation d'IPs[1] pour les ressources fonctionnelles d'un SoC.

[1] pour Intellectual Property

Un langage impératif est par construction séquentiel – une suite d'instructions – et de plus l'habitude de penser séquentiellement la logique d'un programme ne facilite pas la tâche. Programmer de sorte que le parallélisme soit visible est plus approprié et, dans le domaine des applications de traitement de signal pour des systèmes embarqués, la modélisation en utilisant des langages de spécification peut représenter une solution.

Dans cette thèse nous nous intéressons au domaine de traitement intensif de signal pour SoC et notamment aux méthodologies de conception pour les SoC multiprocesseurs. La plupart de ces applications sont par essence parallèles et répétitives et correspondent à une suite de calculs appliqués sur des flots de données généralement multidimensionnels. Des langages de spécification sont disponibles pour l'expression à un niveau haut d'abstraction des concepts essentiels : parallélisme, aspect multidimensionnel, dépendances de données, etc. Des modèles de calculs associés définissent les règles de fonctionnement de la spécification et facilitent le passage à l'exécution. Dans ce sens, les modèles statiques qui permettent une définition statique et une exécution déterministe apportent des propriétés très intéressantes pour des applications avec des contraintes de temps réel.

Un autre aspect intéressant est la modélisation conjointe de l'application, de la plateforme matérielle et du placement. Cela permet l'exploration de l'espace de conception et favorise des techniques d'optimisation itératives guidées par les résultats remontés de la simulation/exécution au niveau de la spécification.

Dans l'équipe DaRT de l'INRIA Lille-Nord Europe où j'ai préparé ma thèse de doctorat, nous nous intéressons à la comodélisation haut-niveau des applications de traitement massivement parallèles placées sur des architectures matérielles embarquées SoC contenant des unités d'exécution parallèles et, à partir de cela, la génération du code pour différentes cibles, travaux intégrés dans l'environnement de conception Gaspard2.

Dans cette démarche, les travaux présentés dans cette thèse se situent au niveau haut de la spécification multidimensionnelle pour le traitement intensif de signal, du modèle de calcul et des optimisations de haut niveau pour l'adaptation de la spécification à l'exécution. Ces aspects se regroupent autour du langage de spécification et du modèle de calcul Array-OL utilisé dans l'équipe au niveau spécification. La modélisation des applications et des architectures parallèles se fait en utilisant la décomposition successive en sous-composants interconnectés où un rôle essentiel est occupé par les structures répétitives pour exprimer des traitements parallèles dans l'application et des architectures répétitives.

Pour pouvoir arriver à générer du code « performant », le premier pas est un langage de spécification adapté au domaine d'application. En tant que caractéristiques essentielles pour le domaine des applications de traitement intensif de signal, on peut citer :

A. l'expressivité pour permettre une spécification facile et complète, qui arrive à saisir les caractéristiques des calculs parallèles ;

B. l'aspect multidimensionnel pour une représentation cohérente des structures de données ;

C. le formalisme visuel : une spécification visuelle est plus appropriée à la modélisation du parallélisme qu'une approche textuelle ;

d. la définition statique pour simplifier la gestion des ressources dans le contexte des systèmes embarqués.

Un modèle de spécification exprime le fonctionnement complet d'une application, en exprimant toutes les dépendances de données, mais une telle spécification ne dit rien relativement à l'exécution de l'application. Tous les ordres d'exécution respectent les dépendances de données sont corrects. Mais quel représente l'ordre optimal ? Ce problème, comme toutes les optimisations multicritères, est très complexe et laborieux.

Dans les compilateurs modernes, les techniques d'optimisation s'appuient sur l'utilisation de transformations de boucles et d'heuristiques implémentant différentes stratégies d'optimisation. Des techniques d'optimisation similaires aux transformations de boucles sont utilisées dans l'environnement Gaspard2, mais ce sont dans notre cas des transformations data parallèles au niveau haut de la spécification. De surcroît, des stratégies d'optimisation itérative peuvent être conçues autour ces transformations, guidées par des paramètres remontants du niveau de la simulation.

CONTRIBUTIONS

Les contributions principales de cette thèse sont les suivantes :

1. proposition des sémantiques d'expression des dépendances interrépétitions dans Array-OL, permettant la construction des boucles dans la modélisation. L'aspect uniforme de ces dépendances a un impact réduit sur le parallélisme de la spécification et, par la composition des dépendances à travers la hiérarchie, des dépendances complexes peuvent être exprimées. En plus, ces dépendances fournissent les constructions nécessaires pour l'expression compacte des architectures répétitives interconnectées ;
2. extension des transformations du haut niveau data parallèle Array-OL. L'ensemble de transformations déjà disponibles a subi des extensions visant soit un ajout de fonctionnalités, soit l'extension pour des cas spéciaux, soit de grouper des transformations élémentaires dans des nouvelles transformations avec de nouvelles propriétés dans une vision orientée vers l'utilisation en pratique ;
3. implémentation des transformations de code dans l'environnement Gaspard2 et le branchement à l'éditeur Papyrus UML pour des modèles UML étendus par le profil standard de l'OMG, MARTE ;
4. validation pratique des transformations et étude de stratégies et d'heuristiques d'enchaînement des transformations. Au moment du passage de la spécification à l'exécution, ces optimisations visent adapter la spécification aux contraints de l'exécution et facilitent l'exploration du placement d'une application sur des architectures parallèles distribuées, par l'exploration de l'espace de conception ;
5. étude de l'interaction entre les transformations Array-OL et les dépendances interrépétition, proposition d'un algorithme pour gérer cette interaction et implémentation de l'algorithme.

PLAN

Dans la première partie de la thèse on s'intéresse aux méthodologies de conception des SoC.

En premier temps, dans le chapitre 1 – MODÉLISATION MULTIDIMENSIONNELLE –, nous faisons le point sur les langages de spécification conçus pour le traitement intensif de signal, avec un intérêt spécial pour l'expressivité des fonctions d'accès régulier.

La sémantique du langage ARRAY-OL et son pouvoir d'expression est exploré dans chapitre 2 – ARRAY-OL – en comparaison avec des langages similaires, suivie par la proposition d'une extension pour pouvoir exprimer des dépendances uniformes entre des répétitions data parallèle.

Dans le chapitre 3, COMODÉLISATION DANS UNE VISION RÉPÉTITIVE EN MARTE, nous allons voir comment les concepts de la décomposition répétitive ARRAY-OL peuvent être utilisés dans la spécification en MARTE des applications de traitement intensif de signal, des architectures répétitives et du placement des spécifications data-parallèles sur les architectures distribuées contenant des unités de calcul parallèles.

Dans la deuxième partie de la thèse, nous allons enquêter sur des techniques de refactoring utilisant des transformations de haut niveau data parallèle, travaux développant les résultats des thèses de deux anciens thésards de l'équipe, Soula et Dumont.

La méthodologie de passage à l'exécution et des techniques d'optimisation sont étudiés dans le chapitre 4 – OPTIMISATIONS SOURCE À SOURCE. Les outils de refactoring basés sur des transformations de haut niveau ARRAY-OL sont approfondis dans le chapitre 5 – REFACTORING EN UTILISANT DES TRANSFORMATIONS DE HAUT NIVEAU – et des extensions sont proposées.

Nous focalisons notre attention sur l'interaction des transformations avec l'extension proposée pour exprimer des dépendances uniformes sur des répétitions data-parallèles dans le chapitre 6 – IMPACT DU REFACTORING SUR LES DÉPENDANCES UNIFORMES – et nous proposons un algorithme qui gère cette interaction.

La dernière partie de cette thèse est orientée vers la validation des travaux théoriques d'optimisation, par l'étude des stratégies d'optimisation dans le contexte d'une application réelle de traitement intensif de signal dans le chapitre 7 – STRATÉGIE DE REFACTORING POUR L'EXPLORATION DE L'ESPACE DE CONCEPTION. Les outils de refactoring dans le contexte de l'environnement GASPARD2 et les travaux d'implémentation, intégration et interfaçage sont présentés dans le chapitre 8, REFACTORING DANS LE CONTEXTE DE GASPARD2 , avant d'en venir aux conclusions.

Première partie

MODÉLISATION

1 MODÉLISATION MULTIDIMENSIONNELLE

Dans ce chapitre nous essayons d'identifier des caractéristiques essentielles du domaine de la modélisation des applications de traitement intensif de signal pour l'embarqué. Nous allons examiner différentes propositions de langages de modélisation pour le domaine, qui ont chacune leurs avantages et leurs faiblesses, en insistant sur le pouvoir d'expression dans le contexte multidimensionnel.

La section 1.1 introduira tout d'abord le domaine du traitement intensif de signal sur System-on-Chic, suivie par une la description des caracteristiques principales des applications multidimensionnelles dans la section 1.2 et une classification des langages dédies au traitement du signal, dans la section 1.3. Les approches fonctionnelles, impératives et flot de données sont étudiées respectivement dans les section 1.4, section 1.5 et section 1.6. Les conclusions sont présentées dans la section 1.7.

1.1 TRAITEMENT INTENSIF DU SIGNAL SUR SYSTEM-ON-CHIP

Le traitement de signal consiste à analyser et interpréter des signaux provenant de capteurs. Nous nous intéressons principalement aux applications de traitement de signal intensif et régulier.

La complexité de ces applications n'est pas causée par les fonctions élémentaires qu'elles combinent, mais par leur enchaînement et de la façon par laquelle l'accès aux tableaux intermédiaires est fait. Une modélisation capable d'exprimer les différentes caractéristiques de telles applications faciliterait l'étape de la spécification et permettrait d'effectuer des nombreuses optimisations.

La charge en calcul de ces applications est assez élevée pour clairement impliquer des unités matérielles de calcul parallèles. Dans cette direction, l'exploitation du parallélisme disponible est essentielle, mais il ne faut pas oublier la performance temps réel qui est un besoin clé de la plupart de ces applications.

Depuis le début des années 2000, les System-on-Chips (ou SoC) ont émergé comme un nouveau paradigme pour la conception de systèmes embarqués. Dans une architecture SoC, plusieurs unités fonctionnelles (processeurs programmables, mémoires, périphériques d'entrée/sortie, etc.) sont intégrés sur la même puce. De plus, de multiples processeurs peuvent être intégrés dans un SoC (Multiprocesseur System-on-Chip, MPSoC [46]) et les communications peuvent être réalisées à travers des réseaux embarqués (Networks-on-Chips, NoCs [11]). Les MPSoCs sont particulièrement avantageuses dans des applications où les contraintes de performances sont cruciales. La Figure 1 illustre un exemple de MPSoC avec des composants interconnectés, l'architecture du TILE64 de Tilera.

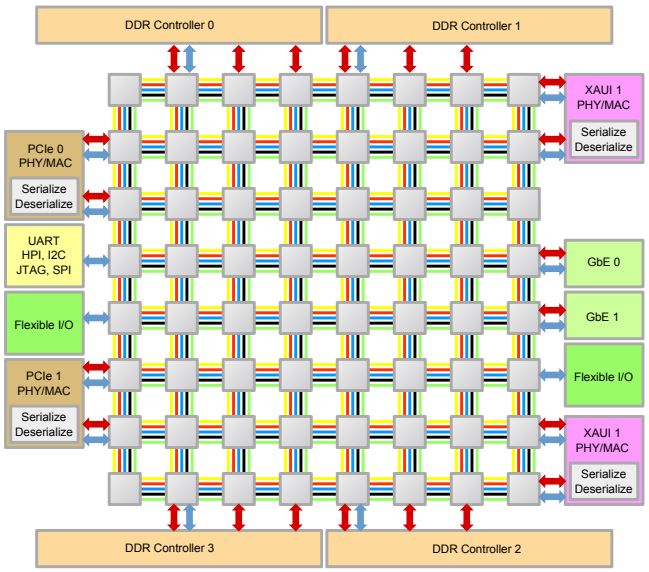


FIG. 1: TILE64

La conception des SoC couvre des points de vue variés y compris la modélisation par l'agrégation des composants fonctionnels, l'assemblage des composants physiques existants, la vérification et la simulation du système modélisé et la synthèse d'un produit final intégré dans une seule puce.

Le placement des applications de traitement intensif de signal sur des SoC et surtout des MPSoC est un problème délicat. Le placement des calculs sur les unités d'exécutions potentiellement parallèles, l'allocation des tableaux dans les mémoires influencent directement les performances. L'évaluation des performances se fait très souvent par la simulation et en remontant des résultats du niveau simulation au niveau spécification et l'exploration des différents placements permet l'augmentation des performances.

La définition statique de la spécification peut réduire les mécanismes de management des ressources embarqués et aussi garantir des latences de fonctionnement temps-réel pour des applications exigeantes.

Nous allons maintenant voir quelles sont les caractéristiques des applications de traitement intensif de signal et comment différents langages de spécification ont approché la thématique.

1.2 APPLICATIONS DE TRAITEMENT INTENSIF MULTIDIMENSIONNEL

Une modélisation de haut-niveau doit permettre d'exprimer correctement et complètement les caractéristiques d'une application.

Des applications de calcul intensif sont prédominantes dans plusieurs domaines d'applications, tels que le traitement d'images et vidéo ou les systèmes de détection (radar, sonar). Par multidimensionnel on entend que ces applications manipulent principalement des structures de données multidimensionnelles représentées normalement par des tableaux. Comme exemple, une vidéo est un objet 3D avec deux dimensions spatiales et une temporelle. Dans une application sonar, une dimension peut être l'échantillonnage temporel des échos, une autre l'énumération des hydrophones et des autres comme les dimensions des fréquences peut apparaître plus tard dans les calculs.

Traiter ce type d'applications présente un nombre de difficultés :

- elles sont d'habitude massivement parallèles et pour bénéficier de ce parallélisme, on doit l'exprimer complètement ;
- seul très peu de modèles de calcul (MoC[1]) sont multidimensionnels et aucun n'est largement utilisé ;
- les motifs d'accès aux tableaux des données sont divers et complexes et cela représente le problème principal quand on parle d'optimisation ;
- l'ordonnancement de ces applications avec des ressources limitées est un défi, spécialement dans un contexte parallèle et distribué.

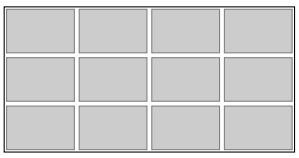
[1] *Model of Computation* en anglais.

On peut retrouver plusieurs essais de conception des langages ou des modèles de calcul dédiés à ce domaine d'applications. Le défi est de proposer une façon pour exprimer des accès multidimensionnels sans compromettre l'utilisation et s'il est possible de permettre un ordonnancement statique de ces applications sur des plateformes matérielles parallèles.

Un bon langage pour des applications de traitement intensif multidimensionnel devrait être capable d'exprimer :

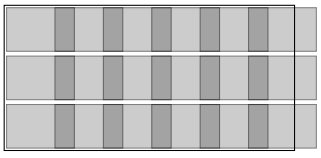
MOTIFS UNIFORMÉMENT ESPACÉS :

Le traitement intensif de signal est extrêmement régulier et il présente beaucoup de parallélisme. Pour exprimer la totalité des caractéristiques de ces applications, il est essentiel de décrire complètement et correctement toute cette régularité.



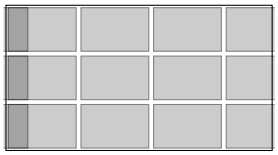
FENÊTRES GLISSANTES :

Les algorithmes en utilisant des fenêtres glissantes sont des parties fondamentales des systèmes de traitement des signaux. Les modèles de flots de données ont des difficultés à exprimer des concepts clefs nécessaires pour exprimer ces algorithmes, tels que la consommation multiple de données ou le traitement des frontières.



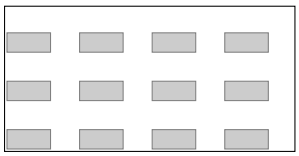
DIMENSIONS CYCLIQUES :

Dans le cas des applications où certaines dimensions spatiales peuvent représenter des tori physiques – comme des hydrophones autour d'un sous-marin –, la capacité de spécifier des dimensions cycliques dans les tableaux est essentielle. Aussi après une transformation Fourier discrète, la dimension de la fréquence est cyclique. Pour gérer les bandes de fréquence, la possibilité d'adressage cyclique d'une plage d'indices est extrêmement importante.



SOUS/SUR ÉCHANTILLONNAGE :

Des contraintes applicatives peuvent causer certains composants à consommer des tableaux qui représentent des sous/sur échantillons des tableaux disponibles.



SPÉCIFICATION HIÉRARCHIQUE : Une spécification hiérarchique est obligatoire pour modéliser des applications complexes. Elle favorise la modularité et la réutilisation des composants, tout en permettant la structuration de l'application aux différents niveaux de granularité.

INFORMATIONS D'ÉTAT : Dans une spécification orientée flot de données où il y a pas des variables globales et les cycles ne sont pas permis dans le graphe, pour permettre de garder des informations d'état – nécessaires pour exprimer des filtres récursifs par exemple – ces structures devraient être exprimées explicitement.

1.3 CLASSIFICATION DES LANGAGES

On peut retrouver plusieurs critères de comparaison pour les langages dédiés au traitement de signal :

FLOT DE DONNÉES, FONCTIONNEL OU IMPÉRATIF ? Le paradigme de flot de données est l'approche largement utilisée dans la description des applications de signal. Une approche fonctionnelle a l'avantage de pouvoir exprimer directement des fonctions d'accès par des expressions mathématiques. Les langages impératifs ont l'atout du pouvoir d'expression dans l'étape de spécification, mais cela complique énormément l'étape de compilation. Des compromis peuvent être atteints en mettant des restrictions sur un tel langage.

VISUEL OU TEXTUEL ? La méthode de saisie des applications influence directement la qualité des modélisations. Certaines catégories de langages possèdent des caractéristiques qui font qu'une représentation visuelle n'est pas seulement meilleure, mais nécessaire. La sémantique de graphe des langages de flot de données fait d'eux des candidats idéaux pour des représentations et manipulations visuelles. Par contre, pour les langages impératifs une représentation textuelle est normalement plus appropriée. Néanmoins, des éléments visuels peuvent aussi être mélangés aux éléments textuels[2].

[2] Une vue visuelle pour le niveau global et des éléments textuels pour des détails.

STATIQUE OU DYNAMIQUE ? Un langage statiquement ordonnançable facilite considérablement les mécanismes au moment de l'exécution, en réduisant les temps d'exécution (pas de réallocation de mémoire, pas d'ordonnancement à la volée, tout est connu avant l'exécution). Il est évident qu'un langage statique est plus restrictif

MoC	Structures de données	Type d'accès	Généralité d'accès			Structures de control
			fenêtres glissantes	sous/sur echantionage	non // avec les axes	délais
SDF [60, 61]	1D	sub-array	-	-		+
CSDF [13]	1D	sub-array	-	-		+
MDSDF [62, 18]	mD	sub-array	-	+	-	+
GMDSDF [70, 69]	mD	sub-array	-	+	+	+
WSDF [47, 50]	mD	sub-array	+	+	-	+
ARRAY-OL [29, 28, 14]	cyclic mD	sub-array	+	+	+	ext.
Stream-IT [87]	1D	sub-array	+	+		+
Alpha [59, 24]	polyhedra	affine	+	+	+	+
Sisal [40, 6]	mD	sub-array	+	+	-	+
SaC [84]	mD	sub-array	+	+	+	+

1. concernant la colonne de types de données, 1D représente des flots de données monodimensionnels (qui peuvent contenir d'autres tableaux comme avec StreamIT), multi-D veut dire que ces flots de données ont été remplaces par des tableaux multidimensionnels, cycliques multi-D représente des tableaux multidimensionnels avec certaines dimensions qui peuvent être cyclique et polyèdres veut dire que le langage manipule des polyèdres convexes de points entiers ;
2. un + dans une colonne signifie que cette particularité est supportée, un - qu'elle n'est pas et ext. qu'elle est supportée par une extension du langage de base.

TAB. 1: Comparaison entre divers modèles de spécification pour le traitement de signal

qu'un langage non statique, de par les contraintes pour assurer qu'un ordonnancement statique peut être calculé.

OBJECTIFS DU LANGAGE. Les caractéristiques d'un langage sont considérablement influencées par le but dans lequel il était conçu : la programmation, la simulation fonctionnelle, l'analyse d'ordonnancement, la synthèse, etc.

Le Tableau 1 présente une comparaison entre plusieurs langages (ou modèles de calculs, MoC) dédiés au traitement de signal. Cette comparaison met en évidence la pertinence de ces divers langages pour la modélisation des applications de traitement de signal multidimensionnels. Les structures de données possibles (flots monodimensionnels ou des tableaux multidimensionnels) et le pouvoir d'expression des fonctions d'accès à ces structures de données comptent parmi les critères principaux de comparaison. La catégorie des applications que ces langages sont capables de traiter est aussi contrainte par les mécanismes de flot de contrôle permis.

1.4 APPROCHE FONCTIONNELLE

La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques et rejette le changement d'état et la mutation des données. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état. L'approche fonctionnelle s'affranchit de façon radicale des effets secondaires en interdisant toute opération d'affectation. Le paradigme fonctionnel n'utilise pas de machine d'états pour décrire un programme, mais un emboîtement de fonctions que l'on peut voir comme des « boîtes noires » que l'on peut imbriquer les unes dans les autres.

1.4.1 *Alpha*

Le langage ALPHA [59, 24], développé au laboratoire Irisa de Rennes, est généralement associé à l'environnement MMALPHA qui est une interface basée sur Mathematica pour la manipulation du langage ALPHA. L'intérêt d'ALPHA est de fournir un langage de haut niveau pour synthétiser des architectures VLSI[3]. En effet, ALPHA est un langage fonctionnel basé sur le modèle polyédral, à assignation unique et fortement typé qui permet de représenter les algorithmes sous forme d'équations récurrentes [49].

[3] VLSI : Very Large Scale Integration caractérise les circuits intégrés de très haute intégration.

ALPHA ne présente pas de particularismes notables concernant la déclaration des fonctions, en revanche les variables sont définies à l'aide de fonctions sur le domaine $\mathbb{Z}^n$. En effet, les données manipulées par ALPHA sont multidimensionnelles : elles correspondent à des unions de polyèdres convexes. Leurs formes ne sont donc pas restreintes à de simples tableaux rectangulaires. L'exemple suivant montre la déclaration d'une variable dont le domaine est l'ensemble des points dans le triangle : $0 \leqslant i \leqslant j; j \leqslant 10$:

```
a : {i,j | 0 ⩽ i ⩽ j; j ⩽ 10}
```

Afin d'accéder aux différentes valeurs de ces données, il est possible de faire des restrictions sur les domaines. On se sert pour cela de l'instruction `case`.

```
 a =
case
{i,j | j = 0 } : 0.(i,j->);
{i,j | j > 0 } : a.(i,j->i,j-1)+1.(i,j->);
esac;
```

Dans cet exemple, nous avons $a[i,j] = 0$ lorsque $j = 0$ et $a[i,j] = a[i,j-1] + 1$.

ALPHA se révèle donc être en mesure d'exprimer de façon simple des formes de données très complexes, mais il s'avère incapable de gérer les accès cycliques. En outre, nous n'avons pas besoin de gérer des formes de données aussi complexes et nous pouvons donc nous contenter de simples tableaux à plusieurs dimensions.

1.4.2 *Sisal*

Sisal (Stream and Iteration in a Single Assignment Language[4]) est un langage fonctionnel à assignation unique conçu dans l'objectif de fournir une interface utilisateur d'usage générale pour une large plage des plateformes parallèles. Ces sémantiques précisent la dynamique d'un graphe de flot de données et les expressions Sisal évaluent et renvoient des valeurs basées uniquement sur les valeurs limitées à leurs arguments formels et identificateurs constitutifs. La définition initiale de Sisal (version 1.2 [40]) est basée sur un modèle utilisant des tableaux monodimensionnels. Dans cette définition, les tableaux peuvent être construits par la concaténation des tableaux des dimensions intérieures pour construire un seul tableau monodimensionnel, qui favorise certaines optimisations comme la vectorisation. Les désavantages de cette approche sont le fait que les tableaux doivent être gardés dans une zone de mémoire continue et que les vecteurs inférieurs doivent avoir la même taille.

[4] Flot et intéraction dans un langage d'assignation unique.

La définition initiale a été élargie par Attali *et al.* dans [6] en permettant des sémantiques multidimensionnelles. Pour exprimer des séquences potentiellement infinies avec des sémantiques non strictes et pour pouvoir exploiter le parallélisme pipeline, des *flots*[5] ont été introduits dans Sisal. Entre autres, dans les sémantiques des tableaux multidimensionnels, les sous-tableaux peuvent avoir des tailles différentes. Différemment que dans la version précédente, cette deuxième favorise des accès aux sous-tableaux dans une manière structurée plutôt qu'élément par élément.

[5] définis par la primitive `stream`.

Des spécifications de placement peuvent être utilisées pour placer des valeurs d'un sous-tableau dans un sous-ensemble géométriquement défini du tableau d'origine, d'exprimer des composants diagonaux en utilisant des notations `dot` ou des placements arbitraires des valeurs quand un tableau est utilisé comme vecteur sous-script. Néanmoins, les placements prédéfinis des sous-tableaux restent parallèles avec les axes ou diagonale et pour pouvoir exprimer des structures plus complexes comme des accès cycliques, le placement des sous-tableaux doit être défini élément par élément.

1.5 APPROCHE IMPÉRATIVE

1.5.1 *StreamIt*

StreamIt [87] est une exception dans les langages « textuels » à flots de données. StreamIt, pour *stream-MIT*, est un langage impératif utilisant la programmation orienté objet, développé au MASSACHUSETTS INSTITUTE OF TECHNOLOGY spécialement pour des systèmes modernes orientés flot de données. Le langage est conçu pour faciliter la programmation des applications complexes de flot de données et l'infrastructure de compilation disponible pour le langage vise un placement efficace sur des architectures matérielles.

Des flots de données mono-dimensionnels avec des éléments qui sont des tableaux ne représentent pas des flots de données multi-dimensionnels adéquates ; dans la sous-section 2.5.2 on détaille cette affirmation.

La vision du projet StreamIt se concentre sur la programmabilité, des optimisations spécifiques au domaine et à l'architecture. Le passage simple et directe de la spécification vers l'implémentation le rend très attractif, mais il est assez limité au niveau de l'expressivité ; il sait manipuler seulement des flot de données mono-dimensionnels.

Un programme StreamIt reprend la syntaxe du langage JAVA, il est d'ailleurs possible d'utiliser un compilateur JAVA sur un code en StreamIt après une phase de traduction. En StreamIt les *filtres* constituent la classe de base et ces filtres sont constituées principalement de deux méthodes :

LA MÉTHODE `work` est la plus importante car elle contient la liste des traitements réalisés par un filtre. Entre outre, c'est dans cette méthode que le filtre peut communiquer avec les autres filtres de l'application. Il utilise pour celà des FIFOS qui sont traités comme des attributs de la classe `filter`. L'interaction avec ces FIFOS se fait à l'aide de trois méthodes `push`, `pop` et `peek` qui sont semblables aux méthodes habituelles de manipulations de FIFOS ;

LA MÉTHODE `init` est utilisée pour initialiser le filtre et les FIFOS. Il est ainsi possible de positionner des valeurs initiales dans les FIFOS avant le début de l'exécution. Mais surtout la méthode permet de positionner le nombre de données qui seront consommées ou produites par `push`, `pop` et `peek` dans la méthode `work`.

Il est ensuite possible de relier ces filtres en utilisant trois méthodes différentes qui représentent chacune une topologie différente : un lien direct, une boucle et une dissociation suivie d'un regroupement.

L'interaction de StreamIt avec le flot de données est donc relativement basique. On peut considérer que StreamIt est le pendant « textuel » du langage « graphique » SDF que nous allons étudier dans la section 1.6 et ceci bien que StreamIt dispose de quelques fonctionnalités supplémentaires.

1.5.2 *SaC*

SaC [84] (Single Assignment C) est un langage qui essaye d'intégrer des tableaux n-dimensionnels avec un temps d'accès constant dans des langages impératifs. Il partage des principes de conception avec Sisal, présenté dans la sous-section 1.4.2, mais introduit des tableaux n-dimensionnels comme structures de données principales.

Un tableau dans SaC est représenté par l'ensemble d'un tableau monodimensionnel qui contient les données et un vecteur de *forme* pour définir sa structure, en spécifiant le nombre d'axes (ou nombre de

dimensions) et le nombre d'éléments (les limites des indices) le long de chaque axe. En utilisant la primitive `reshape` un tableau *n*-dimensionnel peut être défini ou redéfini en changeant sa forme.

Similaire aux autres langages, SaC suggère d'utiliser des *opérations composées sur des tableaux*, qui s'appliquent uniformément sur tous les éléments des tableaux ou des sous-tableaux cohérents. La programmation en utilisant des opérations sur les tableaux peut se faire décidément plus concise, compréhensible, et moins susceptible aux erreurs en comparaison avec en utilisant des nids de boucles qui traversent les éléments avec des valeurs spécifiques de début, fin et pas. Ces opérations impliquent d'habitude des décompositions des tableaux en sous-tableaux et le reassemblage dans une autre forme. Avec SaC, l'approche choisie vise à fournir des constructions du langage suffisamment versatile pour permettre la spécification concise et efficacement exécutable des opérations composés sur les tableaux, basés sur des boucles `WITH-loop`. Ces constructions profitent d'être assez-générales et aussi toutes les optimisations de compilateur concernant la génération du code optimisé peuvent se concentrer sur ces constructions.

1.5.3 *Fortran*

Même si le langage Fortran [45] n'est pas un langage dédié au calcul intensif de signal, nous ne pouvons pas faire de ne le mentionner. Il est un des langages impératifs les plus populaires dans le domaine du calcul haute performance, particulièrement adapté au calcul numérique et scientifique.

Fortran englobe une série de versions, dont chacune a évolué pour ajouter des extensions à la langue tout en conservant généralement la compatibilité avec les versions précédentes. Nous sommes intéressés surtout des constructions d'accès aux tableaux, qui sont apparus avec la norme Fortran 90, en permettant des opérations unitaires au niveau des tableaux (ou des sections de tableaux) et son extension pour le calcul parallèle, High Performance Fortran (HPF) [44], qui ajoute des constructions parallèles (`FORALL`).

Les accès uniformes aux sous-tableaux sont définis par des couple *(début*:*fin*:*pas)* pour chaque dimension du tableau et peuvent spécifier des accès parallèles avec les axes. La construction `WHILE` permet des affectations sur des sections non régulières de tableaux. Mais Fortran est essentiellement un langage impératif et permet la spécification des accès complexes aux données. La difficulté principale est dans la spécification des constructions parallèles : le programmeur doit spécifier lui-même les constructions parallèles, gérer les dépendances de données pour assurer une exécution correcte et spécifier le placement des données sur les processeurs.

HPF met la disposition une gamme large de constructions et implémente un modèle de calcul data-parallèle qui supporte la distribution des calculs sur plusieurs processeurs. Ceci permet la mise en œuvre efficace des architectures de type SIMD et MIMD, mais la complexité des décisions nécessite un programmateur expérimenté pour obtenir un code performant.

1.6 APPROCHE FLOT DE DONNÉES

Un flot de données est un paradigme pour la description des applications du traitement du signal pour l'implémentation sur des architectures matérielles parallèles. Pour l'implémentation concurrente, les tâches sont successivement décomposées en sous-tâches qui ensuite peuvent être automatiquement, semi-automatiquement ou manuellement ordonnancés sur des processeurs parallèles, soit au moment de la compilation (statiquement) soit au moment de l'exécution (dynamiquement). Le concept qui consiste d'automatiquement décomposer un programme séquentiel est très intéressant, mais les techniques d'analyse et d'extraction sont limités ; des programmes impératifs très souvent cachent le parallélisme disponible dans l'application. Si le programmeur fournit cette décomposition comme conséquence naturelle de la méthodologie de programmation, nous devons normalement nous attendre à l'usage plus efficace des ressources concurrentes disponibles.

Synchronous *précise que le langage est statiquement ordonnançable.*

Dans le passé, plusieurs modèles de spécification orientés flot de données ont été proposés, par exemple *Synchronous Data Flow* (SDF)[60, 61], *Cyclo Static Data Flow* (CSDF)[13], *Multidimensional Synchronous Data Flow* (MDSDF)[62, 18], *Generalized Multidimensional Synchronous Data Flow* (GMDSDF)[70, 69], *Windowed Synchronoud Data Flow* (WSDF) [47, 50], *Parameterized Synchronous Data Flow* (PSDF)[12] ou *Cyclo Dynamic Data Flow* (CDDF)[91].

La plupart des langages de flot de données sont conçus autour SDF ou autour son extension multidimensionnelle, MDSDF. SDF a été crée avec le but de pouvoir directement déduire un ordonnancement statique pour permettre la construction des implémentations efficaces. Les autres langages développés à partir de SDF ont essayé d'élargir son expressivité, mais en restant compatible avec SDF et en gardant les propriétés statiques si possible. Des applications décrites en SDF ont l'avantage majeur qu'elles sont définies statiquement, mais les restrictions du langage ainsi que la monodimenisonnalité réduit énormément l'ensemble d'applications modélisables.

On peut retrouver dans la littérature deux directions d'extension du langage :

- l'introduction de la multidimensionnalité explorée avec MDSDF et ses extensions, GMDSDF et WSDF ;
- jouer avec des restrictions, en permettant des sémantiques paramétrées (flot de données paramétré) ou l'introduction des modes de fonctionnement dans la modélisation ; par exemple les changements de fonctionnement cyclique présents avec CSDF.

La deuxième extension va vers une fonctionnalité dynamique. Certains langages, comme CSDF, gardent toujours la propriété de définition statique, mais la plupart ont perdu cette propriété et sont passé à une fonctionnalité dynamique, ce qu'on appelle des langages de flot de données dynamiques.

On s'intéresse dans nos travaux plutôt des langages synchrones permettant un ordonnancement statique.

1.6.1 *SDF : Synchronous DataFlow*

Le modèle SDF (*Synchronous DataFlow*) a été créé et développé par Lee et Messerschmitt à partir de 1986 [60, 61]. Il a été ensuite inté-

gré à PTOLEMY, un environnement de modélisation et de simulation d'applications pour l'embarqué.

En SDF, une application est décrite par un graphe orienté dont chaque nœud consomme et produit des données respectivement sur ses arêtes entrantes et sortantes. Dans PTOLEMY, ces données sont appelées jetons, *tokens*, et les nœuds sont appelés acteurs, *actors*. La Figure 2 montre le graphe d'une application décrite en SDF.

En SDF, les noeuds représentent les calculs et les arrêts les flots des données.

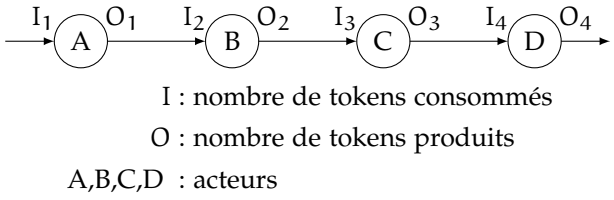


FIG. 2: Une application SDF

Caractéristiques d'une application SDF

Les caractéristiques principales sont :
- le nombre de données consommées et produites par un nœud à chaque exécution est fixé dès la conception de l'application. Ce nombre doit être connu lors de la modélisation. En revanche aucune information n'est nécessaire sur les traitements effectués par ces acteurs ;
- la valeur des jetons ne doit en rien modifier le flot de données. Les données ne doivent servir qu'aux calculs réalisés par les acteurs et ne doivent pas changer le comportement de l'application. Le flot de contrôle est donc indépendant des données.

La définition statique implique des propriétés formelles très intéressantes.

Grâce à ces caractéristiques, l'application est définie statiquement. Il est donc possible d'utiliser les propriétés formelles qui en découlent afin :
- de détecter les interblocages dès la conception ;
- d'ordonnancer l'application dès la modélisation ;
- d'avoir une exécution déterministe (résultats identiques pour quelconque ordonnancement) ;
- d'avoir une exécution dans un temps fini et avec une consommation de mémoire finie.

À l'aide de ces propriétés, SDF permet de tester la validité d'une application dès sa phase de conception, ce qui est bien entendu une grande force.

Calcul de l'ordonnancement

Le calcul de l'ordonnancement se base sur la constatation suivante : lorsque l'application est exécutée, le nombre total de jetons produits par une tâche est égal au nombre de jetons consommés par la tâche suivante. Soit I_x et O_x le nombre de jetons respectivement consommé et produit par la x^e tâche et soit r_x le nombre d'exécution de cette tâche (*cf* la Figure 2), on peut alors écrire l'équation suivante :

$$r_x O_x = r_{x+1} I_{x+1} \tag{1.1}$$

On obtient ainsi un système d'équations ; ce système n'a soit aucune solution, soit une infinité. Dans le premier cas, le graphe est inconsistant et il ne peut être ordonnancé. Dans le deuxième cas, toutes les solutions sont multiples d'une seule et même solution ($\overrightarrow{r}$). L'ensemble des solutions est donc décrit par $k\overrightarrow{r}$ avec $k \in \mathbb{N}^+$.

Lorsque $\overrightarrow{r}$ a été calculé, il est possible, grâce aux résultats présentés dans [60], de construire un graphe de dépendance. Le calcul de $\overrightarrow{r}$ et la construction du graphe de dépendance étant automatiques, il n'y a donc pas de difficultés pour obtenir l'ordonnancement.

Exemple. Soit l'exemple présenté sur la Figure 3 : la première tâche produit deux jetons et la deuxième en consomme trois. La résolution du système d'équation donne $\overrightarrow{r} = \binom{3}{2}$ et le graphe de dépendance obtenu est présenté sur la droite de la figure.

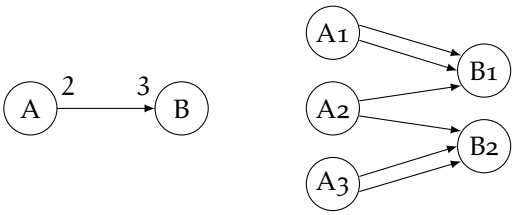


FIG. 3: Graphe de dépendance d'une application SDF

Utilisation des états et des délais

Une autre opportunité offerte par SDF est l'utilisation des « délais ». Un délai est un ensemble de jetons qu'on peut placer sur une arête. Lors de l'exécution de l'application, la tâche, se trouvant en amont de l'arête, produira des jetons, mais ceux-ci ne seront consommés par la tâche en aval qu'après les jetons du délai. Les délais permettent donc de prépositionner des valeurs initiales sur les arêtes.

Un « état » est un délai particulier qui autorise une tâche à utiliser le résultat d'une itération lors de l'itération suivante. Ainsi les n-premiers jetons de l'état sont utilisés par les n-premières itérations de la tâche, puis celle-ci utilise les résultats qu'elle a produits. L'utilisation des états est extrêmement utile : un exemple simple est la somme d'un vecteur où l'état ne comporte qu'un seul jeton de valeur 0.

L'utilisation des états et des délais ne modifie pas la manière de calculer un ordonnancement.

1.6.2 *MDSDF : MultiDimensional Synchronous Dataflow*

Les flots de données où les éléments sont des vecteurs ou des tableaux n'expriment pas une multidimensionnalité appropriée ; les accès sont toujours restreints sur la dimension du flot de données.

SDF autorise la modélisation de flots de données mono-dimensionnelles en spécifiant les dépendances entre les tâches. Les jetons transportés par ces flots de données sont de nature quelconque, il peut notamment s'agir de vecteurs ou de tableaux.

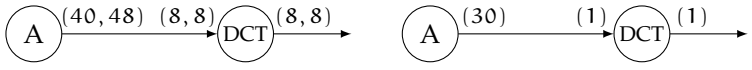
Le principe de MDSDF [62, 18] est de transporter des flots de données multi-dimensionnelles appropriée. Bien que certaines applications MDSDF peuvent toujours être modélisés en SDF par la linéarisation des dimensions, la description est plus compliquée à comprendre et pour la plupart d'applications multidimensionnelles réelles il est impossible de spécifier en SDF comment construire le graphe de précédence.

Le fonctionnement de MDSDF est similaire à celui de SDF. Ainsi, il suffit juste de préciser pour une tâche le nombre de données consommées et produites sur chacune des dimensions du flot. Ce principe est illustré par la Figure 4. Le flot représenté est de forme bidimensionnelle, la première tâche A produit un rectangle de jetons de taille $O_{A,1}$ sur la première dimension et de taille $O_{A,2}$ sur la deuxième pour chaque itération.

FIG. 4: Une application MDSDF

De l'intérêt de MDSDF

L'utilisation de MDSDF permet d'exprimer de manière plus exacte une application. Les figures Subfig. 5a et Subfig. 5b montrent la même application décrite respectivement avec MDSDF et avec SDF. En MDSDF (Subfig. 5a), il est évident que la première tâche produit un tableau de taille 40×48 qui est consommé par morceaux de 8×8 par la deuxième tâche. En SDF il apparaît simplement que la première tâche produit trente jetons et que la deuxième les consomme un par un. SDF ne permet donc pas d'exprimer les dépendances de façon optimale lorsque la structure du flot de données est multi-dimensionnelle.



(a) en MDSDF (b) en SDF

FIG. 5: Une application simple

Mais surtout MDSDF rend possible la modélisation d'application qui ne pouvait être décrite en SDF et ceci même avec une linéarisation des dimensions. La raison en est simple, il n'est pas possible en SDF de spécifier le mode de construction du graphe de dépendances. La Figure 6 montre que la première tâche produit une colonne de deux jetons et que la deuxième tâche consomme une ligne de 3 jetons. Les deux jetons de la colonne ne seront donc pas consommés par la même itération de la deuxième tâche comme le montre le graphe de dépendances. Or la modélisation de cette application en SDF, vue sur la Figure 3 ne respecte pas cette contrainte.

Calcul de l'ordonnancement

Le système d'équation de SDF est remplacé par plusieurs systèmes d'équations. En effet, un système est écrit pour chaque dimension :

$$\begin{aligned} r_{A,1} O_{A,1} &= r_{B,1} I_{B,1} \\ r_{A,2} O_{A,2} &= r_{B,2} I_{B,2} \end{aligned} \tag{1.2}$$

Chaque système est résolu de manière autonome. Les solutions obtenues sont de même forme que celles de SDF, chaque solution s'appliquant

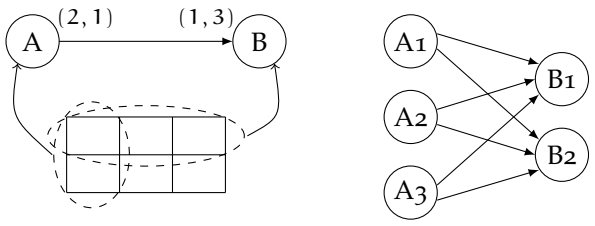


FIG. 6: Une application MDSDF qu'on ne peut transcrire en SDF

à une dimension. Le nombre total d'itérations d'une tâche est égal au produit du nombre d'itérations pour chacune des dimensions.

Particularité du modèle MDSDF

Dans les applications qu'on vient de voir en exemple, le nombre de dimensions du flot de données n'évolue jamais et il n'est d'ailleurs pas possible de créer ou de supprimer des dimensions. C'est pourquoi Lee propose des « acteurs clefs ». Ces derniers ne réalisent aucune opération, ils se contentent juste de modifier la structure du flot de données. D'ailleurs, il est important de rappeler que SDF ne manipule que les flots de données et jamais les données elles-mêmes. La liste des « acteurs clefs » n'est pas prédéfinie, Lee se contente d'en présenter quelques-uns. Le *downsample* sert par exemple à supprimer une dimension. La Figure 7 représente un *downsample* qui consomme un tableau de $M \times N$ et qui produit un vecteur de taille M ; les éléments de la deuxième dimension ont été supprimés.

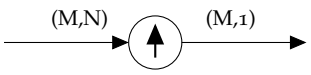


FIG. 7: Un *downsample*

À l'image de SDF, il est possible d'utiliser les « états » et les « délais » en MDSDF. Mais il s'agit désormais de n-uplet qui représente non pas des jetons initiaux mais des lignes et des colonnes de jetons initiaux. Les différences s'arrêtent là et l'utilisation se fait de manière identique à celle de SDF.

Conclusion

MDSDF rend possible l'utilisation de flot de données multidimensionnelles, mais certaines contraintes demeurent. Notamment la consommation et la production des données doivent être parallèles aux axes. Pour résoudre ce problème, Murthy et Lee ont proposé une extension à MDSDF appelé GMDSDF.

1.6.3 *GMDSDF : Generalized MultiDimensional Synchronous Dataflow*

Le but de GMDSDF [70, 69] est de fournir un formalisme capable de modéliser des applications avec une consommation ou une production des données non parallèles aux axes. En GMDSDF, les jetons ne sont plus produits en ligne et en colonne comme avec MDSDF, mais ils sont

placés sur des treillis de points. Seuls certains points du treillis sont consommés ou produits par les tâches de l'application. Trois tâches spéciales sont introduites : la source, le décimateur et l'expandeur, elles sont les seules à pouvoir manipuler les treillis.

Fonctionnement de GMDSDF

Nous introduisons ici les trois tâches spéciales de GMDSDF, puis les tâches ordinaires. Les principes de bases de GMDSDF restent cependant identiques à ceux de MDSDF.

LA SOURCE : elle sert à créer et à définir la forme du treillis. Une source est toujours associée à une matrice exemple *(sampling matrix)* V. Cette matrice détermine la forme du treillis en définissant les points du plan[6] qui en font partie. La matrice treillis est constituée de l'ensemble des points $\overrightarrow{t} = V.\overrightarrow{n}, \forall \overrightarrow{n} \in \mathbb{N}^m, m \in \mathbb{N}$ Sur la partie gauche de la Figure 8, nous constatons que les points du treillis sont générés par la matrice exemple $\left(\begin{smallmatrix} 1 & -1 \\ 1 & 2 \end{smallmatrix}\right)$. De même on peut considérer que pour une application MDSDF bidimensionnelle $V = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$.

[6] Il s'agit bien d'un plan, car comme nous le verrons plus tard, GMDSDF ne s'applique pas lorsque le nombre de dimensions est supérieur à deux.

Une autre matrice est associée avec une source, il s'agit de la matrice support *(support matrix)* W. Elle est utilisée pour déterminer quels points du treillis seront produits. En effet une source ne produit pas tous les points du treillis. La matrice exemple ne sert qu'à donner la forme du treillis et la matrice support, elle, indique l'emplacement des éléments produits par la source. Le calcul de ces positions s'effectue de la manière suivante :

- à partir de la matrice support, il est possible de construire le « parallélépipède fondamental » *(fundamental parallelepiped)* en se basant à l'origine et en prenant les vecteurs de la matrice support comme les deux premiers cotés (*cf* partie droite de la Figure 8) ;
- tous les points à coordonnées entières se trouvant à l'intérieur de ce parallélépipède sont appelés les « points renumérotés » *(renumbered points)* et sont notés $N(W)$; en outre, Lee montre que la cardinalité de cet ensemble est égale à la valeur absolue du déterminant de la matrice support $|N(W)| = |\det W|$;
- la multiplication des coordonnées des « points renumérotés » par la matrice exemple donne les coordonnées des points qui seront effectivement produits par la source (*cf* la Figure 8).

Une source est définie par sa matrice exemple et sa matrice support.

LE DÉCIMATEUR ET L'EXPANDEUR : Si la source permet de créer des treillis, le décimateur et l'expandeur permettent eux d'en modifier la forme : le premier en enlevant des points, le deuxième en en rajoutant. Ils sont respectivement définis par leur matrice de « décimation » (M) et d'« expansion » (L).

Murthy et Lee notent respectivement V_e et V_f les matrices exemples d'entrées et de sorties. De la même facon, il note W_e etW_f les matrices supports. Les liens unissant les treillis d'entrées de sorties sont données par les deux relations suivantes :

$$\begin{array}{lll} \text{décimateur :} & V_f = V_e.M & W_f = M^{-1}.W_e \\ \text{expandeur :} & V_f = V_e.L^{-1} & W_f = L.W_e \ . \end{array} \tag{1.3}$$

Ainsi, un décimateur de matrice $M = \left(\begin{smallmatrix} 3 & 0 \\ 0 & 2 \end{smallmatrix}\right)$, utilisé sur un treillis de forme $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$, ne garde qu'un point sur trois sur la dimension horizontale

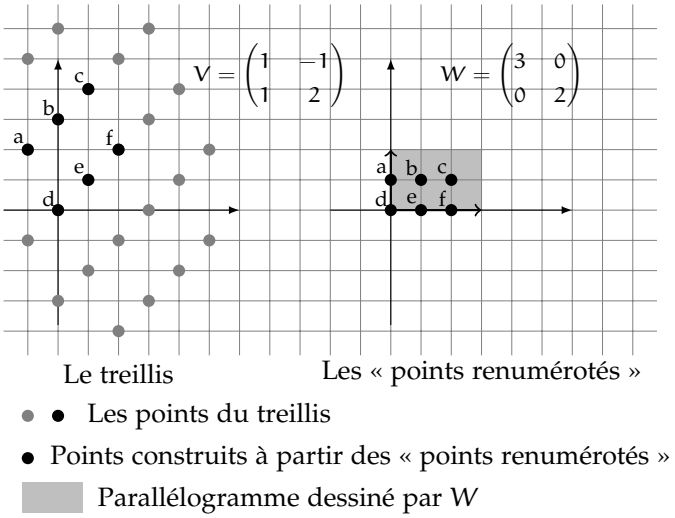


FIG. 8: Fonctionnement d'une source

et un point sur deux sur la verticale. Un expandeur de matrice $L = \left(\begin{smallmatrix} 3 & 0 \\ 0 & 2 \end{smallmatrix}\right)$ permet d'effectuer l'opération inverse.

LES AUTRES TÂCHES : Les tâches « ordinaires » consomment et produisent des points sur des treillis. Cependant elles n'utilisent plus un n-uplet comme en MDSDF mais une matrice support. Ainsi les points désignés ne sont plus nécessairement placés consécutivement sur une ligne ou sur une colonne. Mais ils peuvent être choisis de manière non parallèle aux axes et avec un décalage.

LES ÉTATS ET LES DÉLAIS : Les « états » et les « délais » sont également présents. Comme GMDSDF ignore les notions de lignes et de colonnes, un « état » ou un « délai » sont vus désormais comme un décalage de l'origine lors du calcul du « parallélépipède fondamental ». Le décalage des « points renumérotés » implique également un décalage des points qu'ils désignent. Les valeurs initiales des « états » et des « délais » servent donc à combler ce décalage.

Calcul de l'ordonnancement :

En MDSDF, il est possible de calculer simplement l'ordonnancement en résolvant le système d'équations. Ce système est simple à écrire car les dimensions sur lesquelles sont consommées et produites les données sont parallèles aux axes. Mais en GMDSDF ce n'est pas la cas et si une tâche produit des données suivant un vecteur $\overrightarrow{x}$, la tâche suivante pourra consommer ces mêmes données suivant un vecteur $\overrightarrow{y}$. La désignation du nombre de points est elle aussi différente, tout se fait grâce aux matrices supports qui ne sont pas comparables entre elles. L'établissement du système d'équation est donc beaucoup plus difficile. En fait, Murthy et Lee proposent de ramener le calcul de l'ordonnancement de GMDSDF à celui de MDSDF, ils suggèrent de calculer des boites englobantes pour ranger les points des treillis, mais les calculs sont présentés uniquement pour des applications manipulant des flots bidimensionnels. Les autres types d'applications ne sont pas traités et sont considérés comme trop complexes.

CALCUL DES BOITES ENGLOBANTES : Les données produites ou consommées par chaque tâche sont « rangées » dans des rectangles à l'aide d'une fonction de « rectangularisation ». En effet, il suffit de changer le système de coordonnées d'un treillis en prenant comme base les vecteurs de la matrice exemple[7].

Mais pour le décimateur et l'expandeur, le problème est différent. Il faut seulement que ces derniers ajoutent ou suppriment des ponts sur le treillis ; la quantité de points prise sur un treillis pour une itération importe peu. On fixe alors que lexpandeur consomme un rectangle $(1,1)$ (d'où $W_e = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$) ce qui donne par application de l'Équation 1.3 $W_f = L$. On peut en déduire alors que le décimateur produit un rectangle $(L1, L2)$ avec L1 et L2 deux entiers positifs tels que $L1L2 = |\det(L)|$ [68]. Par un raisonnement similaire le décimateur produit un rectangle $(1,1)$ et consomme un rectangle $(M1, M2)$. Mais on ne peut avoir simplement $M1M2 = |\det(M)|$ car il n'y aurait pas toujours de points à produire en raison de la forme des treillis, il faut donc appliquer d'autres conditions sur le calcul de $(M1, M2)$. En fait, Murthy et Lee prouvent qu'il existe toujours une décomposition de $|\det(L)|$ qui soit valide. Il suffit de tester différentes possibilités jusqu'à ce que l'égalité suivante soit respectée après le calcul de l'ordonnancement :

$$|N(W_f)| = \left|\frac{N(W_e)}{\det(L)}\right| \tag{1.4}$$

CALCUL DE L'ORDONNANCEMENT : Finalement, il est possible de calculer l'ordonnancement de l'application. Une fois ce calcul effectué, il reste pour chaque tâche à définir la matrice support. En effet, on sait grâce aux rectangles combien de données ces tâches consomment, mais on ignore où ces données sont situées sur les treillis. Murthy et Lee prouvent dans [68] qu'il est possible d'obtenir les matrices supports automatiquement à partir du nombre de répétition d'une source. Après ce dernier calcul et la vérification de la validité des boîtes englobantes des décimateurs, la modélisation de l'application et l'ordonnancement sont terminés.

Modélisation d'une application

La modélisation d'une application en GMDSDF s'effectue de la manière suivante :

- création du graphe ;
- choix des matrices exemples, d'« expansion » et de « décimation » pour les tâches spéciales ;
- choix des rectangles pour les tâches « ordinaires » ;
- calcul de l'ordonnancement ;
- calcul des matrices supports.

L'application est donc définie après le calcul de l'ordonnancement, ce qui est un inconvénient majeur puisque tout le travail est effectué par le modeleur.

Conclusion

Il est important de signaler que les extensions introduites dans GMDSDF compliquent beaucoup la résolution du système d'équations nécessaire pour l'ordonnancement statique. Pour calculer cet ordonnancement, plusieurs simplifications sont nécessaires, décrites par Murthy et Lee pour des applications 2D uniquement. Bien que

[7] Les points du treillis sont obtenus par combinaison linéaire de la matrice exemple qui est obligatoirement inversible en GMDSDF ; donc elle est bien une base des points du treillis.

GMDSDF soit une extension du MDSDF, il semble que la meilleure façon de calculer l'ordonnancement est de considérer les applications GMDSDF comme des MDSDF[8].

[8] Les simplifications proposées représentent de réductions aux rectangles généralisés.

Murthy et Lee affirment dans [69] : « la définition de GMDSDF ne serra pas facilement utilisable dans un environnement de développement » et ils n'ont pas implémenté le modèle en PTOLEMY. Ainsi on peut regretter que GMDSDF oblige à définir, à un même niveau de description, les tâches spéciales de manipulation du treillis et les tâches ordinaires de manipulation des données.

1.6.4 *WSDF : Windowed Synchronous Data Flow*

WSDF [47, 50] est un modèle de calcul assez récent, consistant en une évolution de MDSDF et conçu spécialement pour des algorithmes avec des fonctionnalités de *fenêtre glissante*, parties importantes de n'importe quel système de traitement d'images. Les modèles orientés flot de données de la famille SDF ont des difficultés en exprimant des concepts importants des algorithmes de fenêtre glissante ; on peut mentionner l'impossibilité de spécifier des consommations multiples de données ou le traitement des frontières.

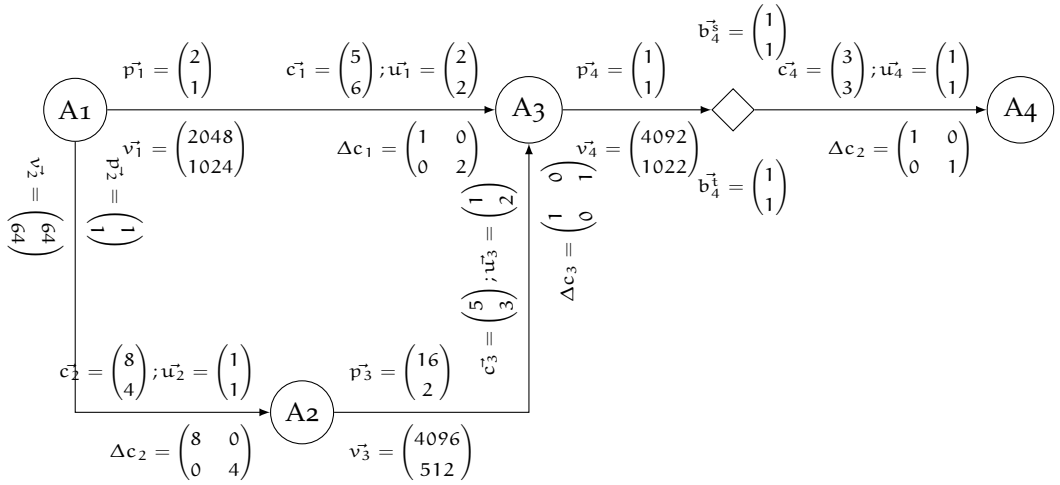
Le modèle WSDF a été créé pour surmonter ces restrictions, en ajoutant des concepts pour l'abstraction des caractéristiques importantes des algorithmes de fenêtre glissante capables de représentation plus précise, mais tout cela en ajoutant un nombre de concepts nouveaux comme des jetons virtuels, des jetons effectifs, unions des jetons virtuels, translation des fenêtres, traitement des frontières, etc. Tous ces concepts additionnels rendent la modélisation plus compliquée et peuvent mener des ambiguïtés. L'avantage du modèle est qu'il garde la propriété de définition statique et il peut être utilisé pour résoudre des questions importantes concernant l'ordonnancement et l'estimation des tampons dans des algorithmes statiques de traitement d'images. Tout cela avec le risque de rendre la spécification trop complexe et en limitant l'applicabilité du modèle aux algorithmes de fenêtres glissantes.

La Figure 9 montre un exemple d'un graphe WSDF, emprunté du [50], utilisé pour illustrer les nouveaux concepts introduits en WSDF. L'idée de WSDF est de permettre la spécification des algorithmes de fenêtres glissantes mais en même temps rester compatible avec MDSDF. La consommation des jetons se fait de la même manière qu'en MDSDF, avec la différence qu'on peut spécifier le décalage de la consommation de ces jetons. Les jetons produits sont groupés en *jetons virtuels* et ensuite en *unions des jetons virtuels*. Keinert *et al.* précisent que la motivation de ces concepts est d'éviter le non-déterminisme. La taille du bloc dans lequel la fenêtre glissante fait ses translations détermine la répétition de l'acteur. En conséquence, si ce bloc de jetons virtuels n'est pas fixe, mais calculé par l'ordonnanceur, l'exécution n'est pas déterministe et peut rendre le modèle ambigu. Alors pourquoi des jetons virtuels et aussi des unions de jetons virtuels ? Ce choix est fait pour rendre plus intuitive la spécification des algorithmes de traitement d'images où plusieurs images sont consommées ensemble, d'après Keinert *et al.*.

Une spécification WSDF avec les concepts additionnels avec des valeurs par défaut est équivalente à une spécification MDSDF.

Les concepts disponibles en WSDF sont :

JETONS EFFECTIFS PRODUITS, $\vec{p}$: concept équivalent avec MDSDF, O, ce sont les jetons produits par l'exécution d'un acteur ;



Acteur *A1* produit deux images de sortie, les deux avec une taille de 2024×1024 pixels, avec la différence que la deuxième est coupée en blocs de 64×64. Acteur *A2* prends cette deuxième image et réalise un sous-échantillonnage sur la dimension verticale et un suréchantillonnage sur celle horizontale, en regroupant les blocs dans une image de taille 4096×512 pixels. Acteur *A3* consommes 2×2 images en provenant de l'acteur *A1*, avec un sous-échantillonnage sur la direction verticale, et 1×2 images de l'acteur *A2*. Les deux images sont consommées avec une fenêtre glissante de dimension 5×6 et respectivement 5×3. Le résultat est passé à l'acteur *A4* qui réalise aussi une opération de fenêtre glissante, mais en utilisant l'extension des frontières.

FIG. 9: Fenêtres glissantes en WSDF

JETONS VIRTUELS, $\vec{v}$: les jetons effectifs sont assemblés dans des jetons virtuels, qui représentent des blocs d'éléments qui forment une unité ;

L'UNION DES JETONS VIRTUELS, $\vec{u}$: représente le support pour l'opération de fenêtre glissante et est composée d'un ou plusieurs jetons virtuels ;

JETONS EFFECTIFS CONSOMMES, $\vec{c}$: concept équivalent avec MDSDF, I, ce sont les jetons consommés à l'exécution d'un acteur. Dans le contexte de WSDF, ceci représente la taille multidimensionnelle de la fenêtre glissante ;

PROGRESSION DE LA FENÊTRE, Δc : une matrice diagonale qui précise la translation de la fenêtre $\vec{c}$ à l'intérieur de l'union des jetons virtuels, $\vec{u}$;

ÉLÉMENTS DE DÉLAI, $\vec{d}$: ont la même interprétation que dans le modèle MDSDF ;

EXTENSIONS DES FRONTIÈRES, $\vec{b^s}$ et $\vec{b^t}$: sans l'extension des frontières, les algorithmes de fenêtre glissante produisent des images plus petites que les images d'entrées, indésirable dans la plupart des cas. $\vec{b^s}$ spécifie combien d'hyperplans sont ajoutés au début de l'union des jetons virtuels, pour chaque dimension, et $\vec{b^t}$ combien d'hyperplans sont ajoutés à la fin de l'union des jetons virtuels, pour chaque dimension. Les éléments des frontières élargis sont censés être des valeurs identiques et constantes.

WSDF modèle est l'extension de MDSDF, donc il englobe toutes les capacités de MDSDF. Des valeurs par défaut pour les concepts additionnels de WSDF donnent un modèle équivalent au MDSDF. Ces valeurs par défaut sont : les jetons produits sont les mêmes que les

jetons virtuels, $\vec{c} = \vec{v}$, l'union de jetons virtuels contient un seul jeton virtuel, $\vec{u} = \vec{1}$, la progression de la fenêtre se fait avec la taille des jetons consommes, $\Delta c = \mathtt{diag}(\mathbf{c})$ et il n'y a pas d'extension des frontières, $\vec{b^s} = \vec{b^t} = \vec{0}$.

1.7 CONCLUSIONS

Nous avons vu dans ce chapitre différents modèles de spécification conçues pour des applications de traitement intensif de signal. Plusieurs approches ont été explorées, résultant de langages variés, chacun avec ses avantages et désavantages. Nous avons insisté sur l'expressivité des langages et sur les caractéristiques statiques.

Avant d'enquêter plus en détail les aspects relatifs à l'expressivité relativement à ARRAY-OL, le langage de spécification et modèle de calcul que nous utilisons, nous allons faire une présentation du langage et de ses sémantiques.

2 ARRAY-OL

Nous avons discuté dans le chapitre précèdent de la modélisation multidimensionnelle pour des applications de traitement intensif de signal et des différentes approches existantes qui essayent d'exprimer et exploiter la régularité disponible dans ce type d'application.

Dans ce chapitre nous allons présenter les principes et la sémantique d'Array-OL, le formalisme que nous utilisons. Suite à une comparaison avec les langages synchrones statiques dérivés de SDF sur le pouvoir d'expression multidimensionnelle, nous identifions une limitation dans le langage en ce qui concerne l'expression des cycles dans l'application et pour la surpasser nous proposons une extension permettant la spécification de dépendances uniformes.

La section 2.1 résume les principes du langage, suivie par la présentation des sémantiques du langage pour l'expression du parallélisme de tâches dans la section 2.2 et de données dans la section 2.3. Les règles de la définition statique du langage sont décrites dans la section 2.4. La section 2.5 s'intéresse au pouvoir d'expression du langage en comparaison avec les autres langages de spécification. Dans la section 2.6 nous introduisons le concept de dépendances uniformes dans la spécification Array-OL, et nous allons finir par des conclusions dans la section 2.7.

2.1 PRINCIPES DU LANGAGE

Tout d'abord, il faut faire la remarque qu'Array-OL est seulement un langage de spécification, il n'y a pas de règles pour l'exécution d'une application décrite avec Array-OL. Néanmoins, un ordonnancement peut être facilement calculé en utilisant cette description comme il est montré dans [14], où l'auteur présente en détail la sémantique formelle d'Array-OL et la définition d'un noyau d'Array-OL permettant un ordonnancement statique.

Le but initial de Array-OL est de fournir un langage mixte graphique-textuel pour modéliser des applications multidimensionnelles de traitement intensif de signal. Comme dit précédemment, ces applications manipulent des tableaux multidimensionnels.

La complexité de ces applications n'est pas causée par les fonctions élémentaires qu'elles combinent, mais par leur enchaînement et de la façon par laquelle l'accès aux tableaux intermédiaires est fait. La difficulté et la variété de ces applications de traitement intensif de signal est en lien direct avec la façon dont ces fonctions élémentaires accèdent aux entrées et aux sorties, en tant que parties des tableaux multidimensionnels. Les motifs complexes d'accès mènent aux difficultés d'ordonnancement efficace sur des plateformes d'exécution parallèles et distribuées.

Étant donné que ces applications traitent des quantités énormes de données sous des contraintes de temps-réel extrêmement serrées, l'utilisation efficace du parallélisme potentiel de l'application sur les structures matérielles parallèles est obligatoire.

Les motifs d'accès aux tableaux multidimensionnels donnent la complexité des applications de traitement intensif de signal.

De ces exigences, on peut formuler les principes de base qui sous-tendent le langage :

A. tout le parallélisme potentiel de l'application doit être disponible dans la spécification, à la fois le *parallélisme de tâches* et le *parallélisme de données* ;

B. Array-OL est un langage d'*expression des dépendances* de données. Seules les vraies dépendances de données sont spécifiées, afin d'exprimer le parallélisme complet de l'application, précisant un ordre partiel minimal pour les tâches. Ainsi, tout ordonnancement respectant ces dépendances va produire le même résultat : le langage est déterministe ;

C. il s'agit d'un formalisme d'*assignation unique*. Aucune donnée n'est écrite deux fois. Par contre, elle peut être lue plusieurs fois. Array-OL peut être considéré comme un langage fonctionnel du premier ordre ;

D. les accès aux données sont faits par des sous-tableaux, appelés *motifs* ;

E. le langage est hiérarchique afin de permettre des descriptions aux différents niveaux de granularité et de gérer la complexité des applications. Les dépendances de données exprimées à un certain niveau (entre les tableaux) sont des approximations des dépendances précises des sous-niveaux (entre les motifs) ;

F. les dimensions spatiales et temporelles sont traitées de la même façon dans les tableaux. En particulier, le temps est expansé comme une dimension (ou plusieurs) des tableaux. C'est une conséquence directe de l'assignation unique ;

G. les tableaux sont vus comme des tores. En effet, certaines dimensions spatiales peuvent représenter des tores physiques (pensez aux hydrophores autour un sous-marin). Les domaines des fréquences obtenues suite aux transformations Fourier discrètes sont aussi toriques.

La sémantique ARRAY-OL *est celle d'un langage fonctionnel de premier ordre qui manipule des tableaux multidimensionnels.*

ARRAY-OL n'est pas un langage de flot de données, mais peut être projeté sur ce type de langage. Le langage ne manipule pas des flots, mais des tableaux des données. L'environnement ou la plateforme d'exécution peut imposer un ordre et une granularité sur les éléments des tableaux et ainsi diriger vers des flots, mais le choix de la granularité des calculs est laissé pour le compilateur (ou l'ingénieur système) et pas le modeleur.

Exemple. Avec la même spécification ARRAY-OL, une vidéo représentée sous la forme d'un tableau 3D de pixels peut être traitée comme un flot d'images, un flot de lignes ou bien même un flot de pixels).

Comme hypothèse simplificatrice, le domaine d'applications qu'ARRAY-OL couvre est restreint. Un contrôle complexe n'est exprimable et le contrôle est indépendant des valeurs de données. C'est un choix réaliste pour le domaine d'application donné, qui est principalement flot de données. Des efforts de couplage des flots de contrôle et des flots de données, dans le cadre d'ARRAY-OL, ont été réalisés par Labbani *et al.* dans [56].

Le modèle usuel pour des spécifications basées sur des algorithmes des dépendances est le graphe de dépendance où les nœuds représentent les tâches, et les arcs les dépendances. Différentes variations de ces graphes ont été définies. Les graphes étendus représentent le parallélisme de tâches disponible dans l'application. Afin de gérer des applications complexes, une extension courante est la hiérarchie. Un nœud peut lui-même être un graphe. ARRAY-OL s'appuie sur tels graphes de dépendances hiérarchiques et ajoute les nœuds répétition pour exprimer le parallélisme de données de l'application.

Formellement, une application ARRAY-OL est une série de *tâches* connectées par des *ports*. Les tâches sont équivalentes à des fonctions mathématiques qui lisent des données sur leurs ports d'entrées et écrivent des données sur leurs ports de sorties. Les tâches sont de trois types : *élémentaire*, *composée* et *répétition* :

A. une tâche élémentaire est atomique (une boîte noire) et elle peut faire partie d'une bibliothèque par exemple ;

B. une tâche composée est un graphe de dépendances dont les sous-tâches sont connectées via leurs ports ;

C. une répétition est une tâche qui exprime comment une seule sous-tâche est répétée.

Toutes les données échangées entre les tâches sont des tableaux. Ces tableaux sont multidimensionnels et sont caractérisé par leur *forme*, le nombre d'éléments sur chacune de leurs dimensions[1]. Une forme va être notée par un vecteur colonne ou une série de valeurs séparées par des virgules indifféremment. Chaque port est caractérisé par la forme et le type des éléments du tableau qu'il lit ou écrit.

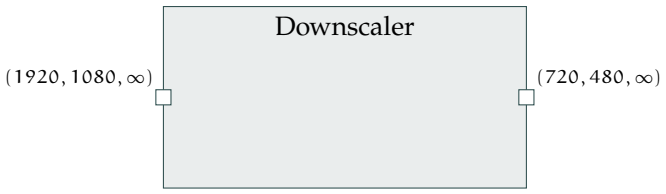
[1] Un seul point, considéré comme un tableau 0-dimensionnel a la forme $()$, considéré comme un tableau 1-dimensionnel à la forme (1), considéré comme un tableau 2-dimensionnel a la forme $\binom{1}{1}$, etc.

Comme indiqué précédemment, le modèle ARRAY-OL est à assignation unique. Il manipule des *valeurs* et pas des *variables*. Le temps est donc représenté comme une (ou plusieurs) dimension(s) dans les tableaux de données. Par exemple, un tableau qui représente une vidéo

est tridimensionnel avec une forme (largeur de l'image, hauteur de l'image, nombre d'images[2].

[2] Un flot vidéo haute-définition infini a une forme $(1920, 1080, \infty)$.

Le reste de la présentation d'ARRAY-OL va être illustré par une application qui met à l'échelle un signal télévision haute définition vers un signal télévision de définition standard (application appelle « Downscaler »). L'illustration graphique de cette application est montrée dans la Figure 10. Les deux signaux seront représentés par des tableaux tridimensionnels.



Les tâches sont représentées par des rectangles nommés, leurs ports par des carrées sur la frontière des tâches. Pour mieux visualiser les flots de données, les ports d'entrées sont placés sur la côte gauche et les ports de sortie sur la côte droite, en exprimant des flots qui vont de la gauche vers la droite.

FIG. 10: Downscaler - boîte noire

Dans la spécification d'une application, une stratégie possible de modélisation est la décomposition hiérarchique de haut en bas. On va suivre cette approche sur l'exemple du Downscaler pour mieux illustrer les concepts ARRAY-OL. À chaque niveau de la hiérarchie, une tâche peut être vue comme une boite noire où sa fonctionnalité peut être explicitée par la décomposition en sous-tâches.

2.2 PARALLÉLISME DE TÂCHES

Le parallélisme de tâches est décrit par une tâche composée. La description composée est un *graphe orienté acyclique*. Chaque nœud représente une tâche et chaque lien une dépendance entre deux ports conformes (même type et même forme). Il n'y a pas de relation entre les formes des ports d'entrée et de sortie d'une tâche ; une tâche peut par exemple lire deux tableaux bidimensionnels et écrire un tableau tridimensionnel. La création ou la suppression des dimensions par une tâche est extrêmement utile, un très simple exemple est la FFT qui crée une dimension de fréquence.

Dans l'application Downscaler de la Figure 10, la fonctionnalité peut être décomposée en deux filtres successifs, le premier qui réduit la dimension horizontale et le deuxième qui ensuite réduit la dimension verticale de chaque image. Cette fonctionnalité est exprimée en utilisant la décomposition en tâche composée, illustrée dans la Figure 11. Les deux filtres sont des sous-tâches et la dépendance entre les deux est représentée par des flèches.

Le graphe est un graphe de dépendances et pas un graphe de flot de données.

Chaque exécution d'une tâche lit un tableau entier sur chaque de ces ports d'entrée et écrit les tableaux entiers sur les ports de sortie.

Alors, il est possible d'ordonnancer l'exécution des tâches uniquement avec la description composée, mais il n'est pas possible d'exprimer le parallélisme de données de nos applications parce que les détails des calculs réalisés par une tâche sont cachés à ce niveau de spécification.

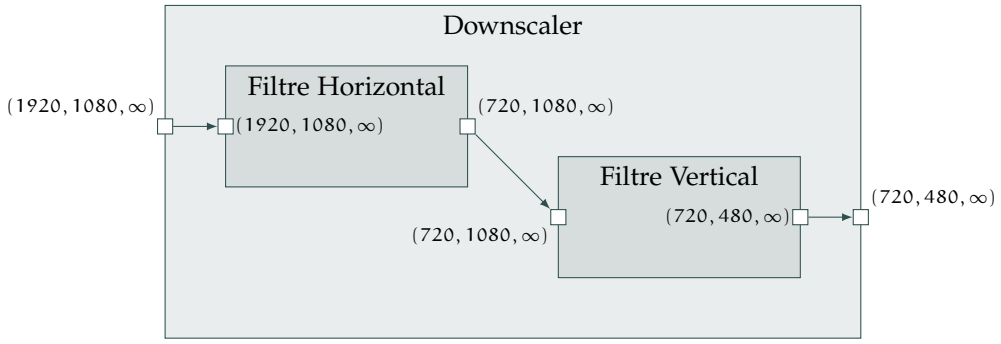


FIG. 11: Downscaler - parallélisme de tâches

2.3 PARALLÉLISME DE DONNÉES

Une répétition data-parallèle d'une tâche est spécifiée dans une tâche répétée. Deux hypothèses sont à la base de ces répétitions :

LES RÉPÉTITIONS D'UNE TÂCHE RÉPÉTÉE SONT INDÉPENDANTES. Elles peuvent être ordonnancées dans un ordre arbitraire et, plus important, même en parallèle[3].

MOTIFS UNIFORMÉMENT ESPACÉS. Chaque instance de la tâche répétée travaille avec des sous-tableaux des entrées et des sorties de la répétition. Pour une certaine entrée ou sortie, toutes les instances des sous-tableaux ont la même forme, sont composées d'éléments uniformément espacés et sont uniformément placées dans le tableau. Cette hypothèse permet une représentation compacte de la répétition et elle est cohérente avec le domaine d'application d'ARRAY-OL qui décrit des algorithmes très réguliers.

[3] C'est pourquoi nous parlons de *répétitions* et ne pas d'*itérations*.

Comme tous ces sous-tableaux sont conformes, ils sont appelés *motifs* quand considérés comme tableaux d'entrées/sorties des répétitions de la tâche répétée et *tuiles* quand considérés comme une série d'éléments des tableaux de la répétition. Pour donner toutes les informations nécessaires à la création de ces motifs, un « tiler »[4] est associé à chaque lien.

Un tiler exprime comment construire les motifs à partir d'un tableau d'entrée, ou comment ranger les motifs dans un tableau de sortie. Il décrit le lien entre les coordonnées des éléments des tuiles et les coordonnées des éléments des motifs. Il contient les informations suivantes :

- F, une matrice d'*ajustage*[5] ;
- **o**, l'*origine* vecteur du *motif de référence* (pour la *répétition de référence*) ;
- P, une matrice de *pavage*.

La répétition de référence est la répétition avec l'indice nul.

[4] En anglais, *tuile* se traduit par *tile* et *tiler* veut dire le constructeur des tuiles ; dans notre cas, la construction mathématique qui permet de spécification compacte des tuiles.

[5] *Fitting* en anglais.

2.3.1 *Représentation visuelle d'une tâche répétée*

Les formes des tableaux et des motifs sont, de la même façon que dans la description composée, notées sur les ports. L'*espace de répétition*, indiquant le nombre des répétitions est défini lui-même comme un tableau multidimensionnel, avec une forme associée. Chaque dimension de l'espace de répétition peut être regardée comme une boucle parallèle et la forme de cet espace de répétition donne les limites des indices du nid de boucles parallèles.

Un exemple de la description visuelle d'une répétition est donnée dans la Figure 12 où on retrouve la répétition du filtre horizontal de l'application Downscaler. Les tilers sont associés aux dépendances qui font le lien entre les tableaux et les motifs. Le filtre horizontal présente un comportement idéal pour être représenté avec des tilers : le même calcul élémentaire, le calcul de 3 pixels à partir d'une fenêtre de 13 pixels est répété sur chaque ligne de chaque image, avec un glissement de 8 pixels pour les motifs d'entrée et avec des motifs qui « collent » un après l'autre pour les motifs de sortie[6]. La construction de ces tilers est détaillée par la suite.

[6] On retrouve un algorithme de fenêtre glissante usuel pour le traitement d'images. À la sortie, l'assignation unique du langage nous oblige de produire des motifs sans chevauchement.

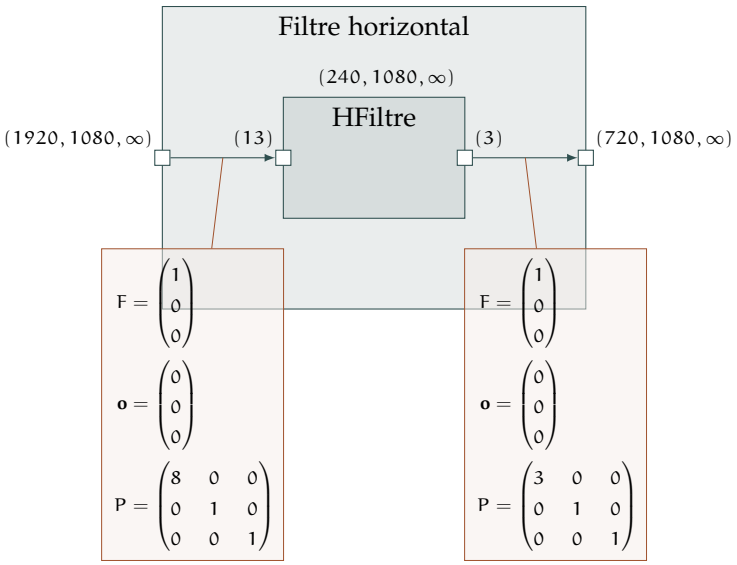


FIG. 12: La répétition du filtre horizontal

2.3.2 *Construire une tuile à partir d'un motif*

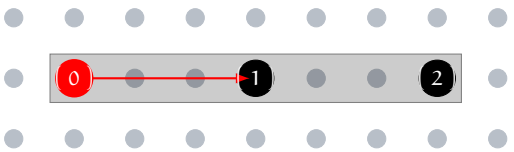
A partir d'un *élément de référence* (**réf**) dans un tableau, on peut extraire un motif en énumérant ses autres éléments relativement à l'élément de référence. La matrice d'*ajustage* est utilisée pour calculer les autres éléments. Les coordonnées des éléments du motif ($\mathbf{e_i}$) sont construites comme somme des coordonnées de l'élément de référence et une combinaison linéaire des vecteurs d'ajustage, comme suit :

$$\forall \mathbf{i}, \mathbf{0} \leqslant \mathbf{i} < \mathbf{s}_{\text{motif}}, \mathbf{e_i} = \mathbf{réf} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{tableau}}, \tag{2.1}$$

où $\mathbf{s}_{\text{motif}}$ est la forme du motif, $\mathbf{s}_{\text{tableau}}$ est la forme du tableau et F la matrice d'ajustage.

Dans les exemples suivants de matrices d'ajustage et tuiles, Figure 13 à Figure 18, les tuiles sont tirées depuis un élément de référence dans un tableau 2D. Les éléments du tableau sont marqués par leur indice dans le motif, $\mathbf{i}$, en illustrant la formule $\forall \mathbf{i}, \mathbf{0} \leqslant \mathbf{i} < \mathbf{s}_{\text{motif}}, \mathbf{e_i} = \mathbf{réf} + F \cdot \mathbf{i}$. Les vecteurs d'ajustage constituant la base de la tuile sont tirés depuis le point de référence.

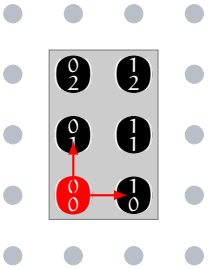
Un élément clef qu'on doit retenir quand on utilise ARRAY-OL est que toutes les dimensions des tableaux sont toroïdales. Cela veut dire que les coordonnées des éléments des tuiles sont calculées modulo la taille



$$F = \begin{pmatrix} 3 \\ 0 \end{pmatrix} \quad \mathbf{s}_{\text{pattern}} = \begin{pmatrix} 3 \end{pmatrix}$$

On a 3 éléments dans cette tuile parce que la forme du motif est de (3). Les indices de ces éléments sont donc (0), (1) and (2). Leur position dans la tuile relativement au point de référence sont donc $F \cdot (0) = \binom{0}{0}$, $F \cdot (1) = \binom{3}{0}$, $F \cdot (2) = \binom{6}{0}$.

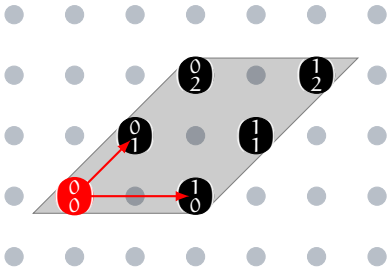
FIG. 13: Tuile éparse



$$F = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \mathbf{s}_{\text{pattern}} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

Le motif est ici bidimensionnel avec 6 éléments. La matrice d'ajustage construit un rectangle tuilé compact dans le tableau.

FIG. 14: Tuile rectangulaire

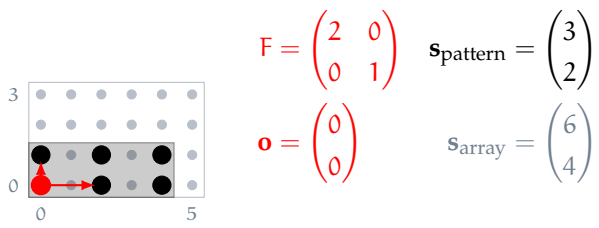


$$F = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \quad \mathbf{s}_{\text{pattern}} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

Cet exemple illustre comment une tuile peut être éparse, à cause du vecteur d'ajustage $\binom{2}{0}$, et non parallèle avec les axes, à cause du vecteur d'ajustage $\binom{1}{1}$.

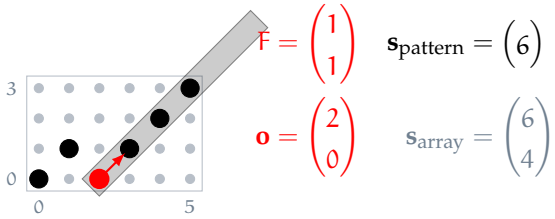
FIG. 15: Tuile complexe (éparse sur une dimension et diagonale sur l'autre)

des dimensions du tableau. Les exemples plus complexes qui suivent sont construits depuis un élément de référence fixe ($\mathbf{o}$ comme origine dans la figure) dans des tableaux de taille fixe, en illustrant la formule $\forall \mathbf{i}, \mathbf{0} \leqslant \mathbf{i} < \mathbf{s}_{\text{motif}}, \mathbf{e}_{\mathbf{i}} = \mathbf{o} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{tableau}}$.



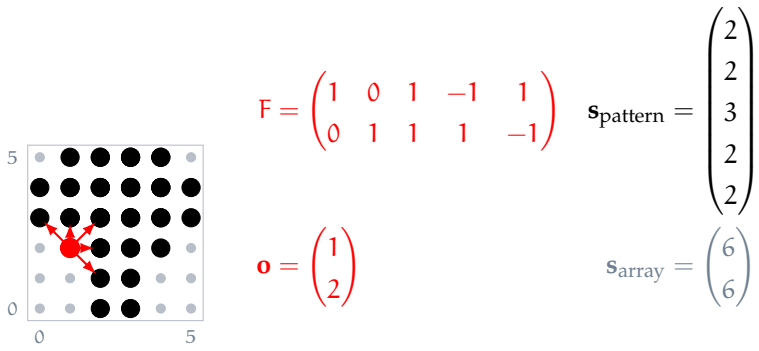
Cet exemple montre la construction d'un motif qui sous-échantillonne le tableau sur la dimension horizontale. Seuls les éléments des colonnes impaires sont utilisés.

Fig. 16: Une tuile éparse, alignée sur les axes du tableau



Le motif est ici monodimensionnel, la matrice d'ajustage construit une tuile diagonale qui s'enveloppe autour le tableau à cause du modulo.

Fig. 17: L'utilisation du modulo



Ceci est un cas extrême d'un motif cinq dimensionnel, ajusté comme une tuile deux dimensionnelle. La plupart des éléments de la tuile sont lus plusieurs fois pour construire le motif de 48 éléments.

Fig. 18: Tuile au motif élargi

2.3.3 *Paver un tableau avec des tuiles*

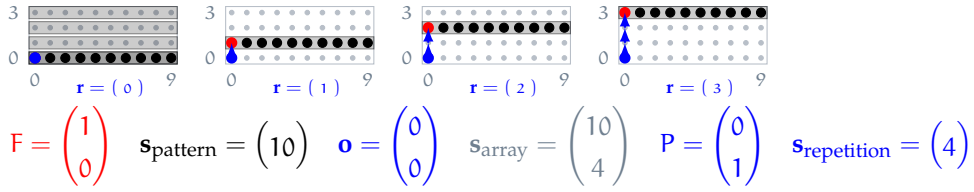
Pour chaque répétition, on a besoin de désigner les éléments de référence des motifs d'entrée et de sortie. Un chemin similaire à celui qu'on a utilisé pour énumérer les éléments d'un motif est employé.

L'élément de la répétition de référence est donné par le vecteur d'*origine*, $\mathbf{o}$, pour chaque tiler. Les éléments de référence des autres répétitions sont construits relativement à celui-ci. Comme auparavant, leurs coordonnées sont construites par une combinaison linéaire des vecteurs de la matrice de *pavage*, comme suit

$$\forall \mathbf{r}, \mathbf{0} \leqslant \mathbf{r} < \mathbf{s}_{\text{répétition}}, \mathbf{réf}_{\mathbf{r}} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{tableau}}, \tag{2.2}$$

où $\mathbf{s}_{\text{répétition}}$ est la forme de l'espace de répétition, P la matrice de pavage, et $\mathbf{s}_{\text{tableau}}$ la forme du tableau.

Quelques exemples sont présentés par la suite, de la Figure 19 à la Figure 2.3.3.



Cette figure représente les tuiles pour toutes les répétitions de l'espace de répétition indexée par **r**. Les vecteurs de pavage tracés depuis l'origine **o** indiquent comment les coordonnées de l'élément de référence $\mathbf{réf}_{\mathbf{r}}$ de la tuile courante est calculée. Ici le tableau est tuilé ligne par ligne.

FIG. 19: Pavage par ligne

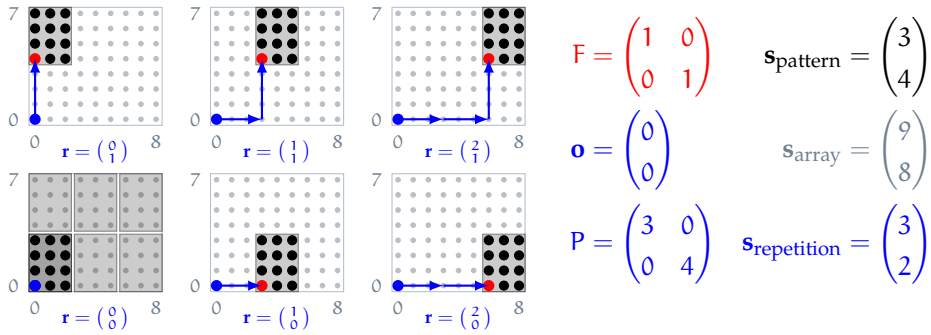


FIG. 20: Un motif 2D qui pave exactement un tableau 2D

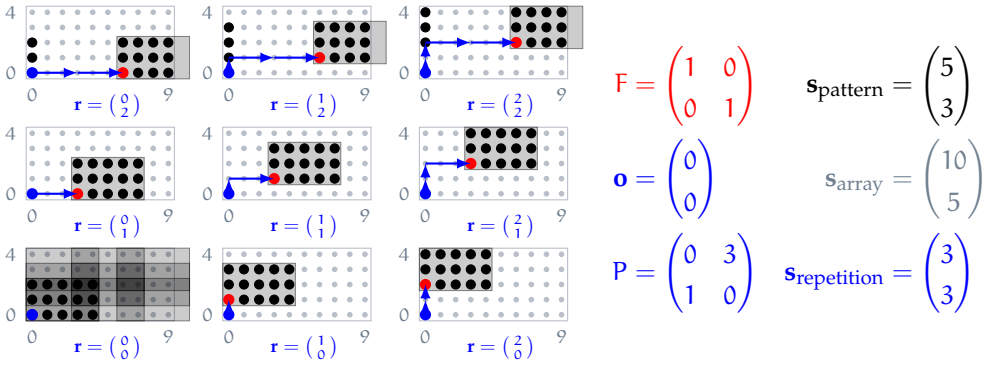


FIG. 21: Les tuiles peuvent se chevaucher et le tableau est toroïdal

2.3.4 *Résumé*

On peut faire le sommaire de ces explications avec une seule formule. Pour un indice de répétition donné $\mathbf{r}$, $\mathbf{0} \leqslant \mathbf{r} < \mathbf{s}_{\text{répétition}}$ et un indice du

motif $\mathbf{i}, \mathbf{0} \leqslant \mathbf{i} < \mathbf{s}_{\text{motif}}$, l'élément qui correspond dans le tableau a les coordonnées

$$\mathbf{o} + (\mathsf{P}\ \mathsf{F}) \cdot \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s}_{\text{tableau}}, \tag{2.3}$$

où $\mathbf{s}_{\text{tableau}}$ est la forme du tableau, $\mathbf{s}_{\text{motif}}$ est la forme du motif, $\mathbf{s}_{\text{répétition}}$ est la forme de l'espace de répétition, $\mathbf{o}$ contient les coordonnées de l'élément de référence du motif de référence, appelé aussi origine, P est la matrice de pavage dont les vecteurs colonne, appelés vecteurs de pavage, représentent l'espacement uniforme entre les motifs, F est la matrice d'ajustage dont les vecteurs colonne, appelés vecteurs d'ajustage, représentent l'espacement uniforme entre les éléments d'un motif dans le tableau.

Des contraintes sur les dimensions des vecteurs et des matrices peuvent être tirés de leur utilisation :

- l'origine, la matrice d'ajustage et celle de pavage ont le même nombre de lignes, égal au nombre de dimensions du tableau ;
- la matrice d'ajustage a un nombre de colonnes égal au nombre de dimensions du motif[7] ;
- la matrice de pavage a un nombre de colonnes égal au nombre de dimensions de l'espace de répétition.

[7] Ainsi, si le motif est un élément seul vu comme un tableau 0-dimensionnel, la matrice d'ajustage est vide et notée par $()$. Le seul élément d'une tuile est donc son élément de référence. On peut regarder cela comme un cas dégénéré de l'equation d'ajustage, où l'index $\mathbf{i}$ n'existe pas et ainsi pas de multiplication $\mathsf{F} \cdot \mathbf{i}$.

2.3.5 *Faire le lien entre les entrées et les sorties via l'espace de répétition*

Les formules précédentes décrivent quels éléments d'un tableau d'entrée ou de sortie sont consommés ou produits par une répétition. Le lien entre les entrées et les sorties est fait par l'indice de répétition, $\mathbf{r}$. Pour une répétition donnée, les motifs de sorties (de l'indice $\mathbf{r}$) sont produits par la tâche répétée depuis les motifs d'entrées (de l'indice $\mathbf{r}$). Les éléments de ces motifs correspondent aux éléments des tableaux via les tuiles associées à ces motifs. Ainsi, l'ensemble des tilers et les formes des motifs et de l'espace de répétition définissent les dépendances entre les éléments des tableaux de sortie et les tableaux d'entrée d'une répétition. Comme indiqué précédemment, aucun ordre d'exécution n'est défini implicitement par ces dépendances entre les répétitions.

Pour illustrer ce lien entre les entrées et les sorties, la Figure 22 présente plusieurs répétitions du filtre horizontal de l'application Downscaler. Pour simplifier la figure et vu que le traitement se fait image par image, seules les deux premières dimensions des tableaux sont représentées[8]. Les tailles des tableaux ont été réduites aussi par un facteur de 60 sur chaque dimension pour raisons de lisibilité.

[8] Effectivement, la troisième dimension des tableaux d'entrée et de sortie est infinie, la troisième dimension de l'espace de répétition est aussi infinie, les tuiles ne croisent pas cette dimension et le seul vecteur de pavage ayant un troisième élément non nul est $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ le long de la dimension infinie de l'espace de répétition.

2.3.6 *Exemples d'accès uniformes*

Le formalisme d'Array-OL permet la spécification d'un éventail vaste des motifs d'accès, qui peut varier de simples motifs tuilés par blocs à des accès complexes où des tuiles non parallèles avec les axes sont associées avec des pas et du modulo. Pourtant, dans les applications de traitement intensif de signal, la grande majorité des accès restent parallèles avec les axes ou utilisant des fenêtres glissantes.

Les constructions les plus usuelles sont :

ACCÈS PARALLÈLE AVEC LES AXES. Avec ce type de construction, les vecteurs des matrices de pavage et d'ajustage sont tous par-

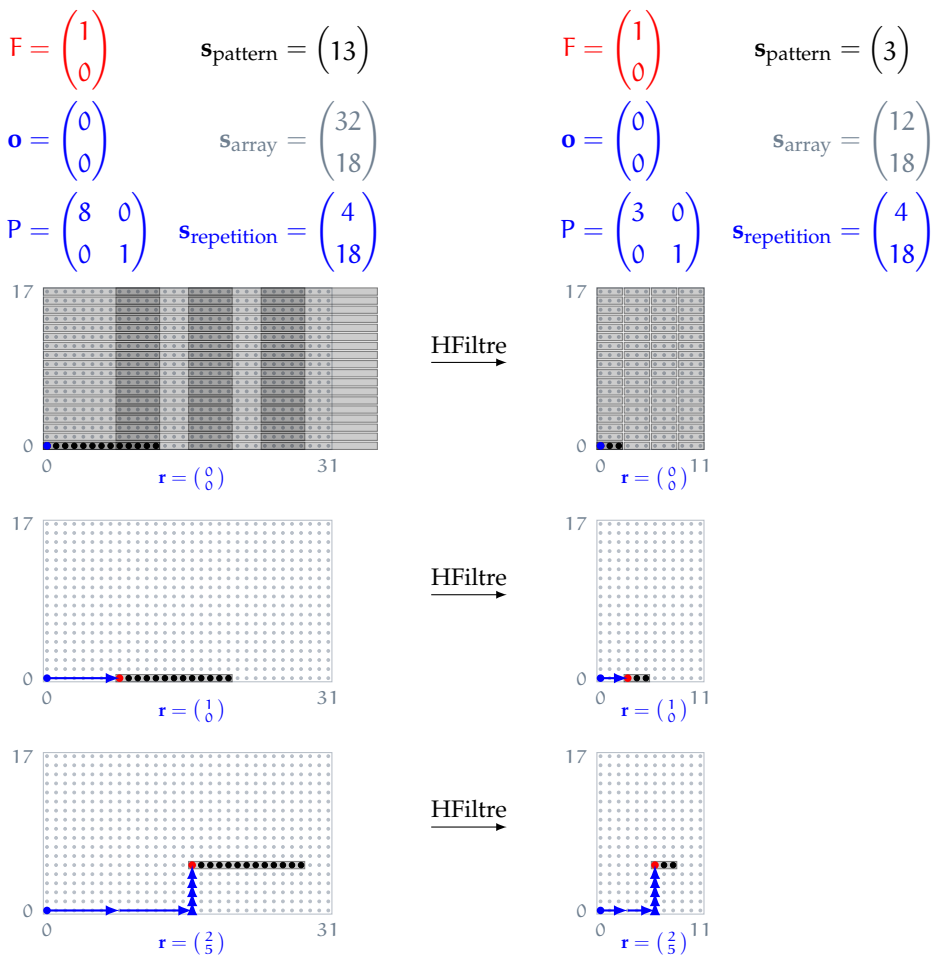


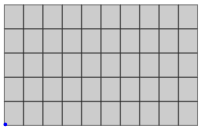
FIG. 22: L'enchaînement des entrées et des sorties de filtre horizontal

allèles avec les dimensions du tableau. Chaque vecteur contient au plus une valeur non-nulle et l'index de cette valeur fait la correspondance entre la dimension du motif ou de la répétition associée au vecteur et la dimension du tableau ;

PAVAGE PAR BLOCS. Des motifs rectangulaires et tuilés parfaitement sont accédés. Le matrices de pavage et d'ajustage expriment des constructions parallèles avec les axes où les motifs sont de blocs continus du tableau, (chaque vecteur d'ajustage contient une seule valeur égale à 1[9]) et les blocs sont parfaitement tuilés un après l'autre dans le tableau (les vecteurs de pavage ont des valeurs non-nulles égales avec la dimension du motif avec laquelle le vecteur partage la dimension du tableau) ;

Exemple.

Un tableau avec une forme de $(80, 50)$ peut être tuilé en $(10, 5)$ blocs de taille $(10, 8)$, en utilisant une matrice d'ajustage de $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ et une matrice de pavage de $\left(\begin{smallmatrix} 8 & 0 \\ 0 & 10 \end{smallmatrix}\right)$, comme illustrée sur la figure à droite.

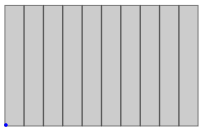


[9] Une matrice pseudo-identité, sur chaque ligne ou colonne on peut retrouver au plus une valeur de 1, sans que la matrice soit obligatoirement carrée.

DIMENSIONS ENTIÈREMENT PAVÉES. Un cas spécial d'accès par bloc où certaines dimensions des motifs correspondent aux dimensions entières du tableau.

Exemple.

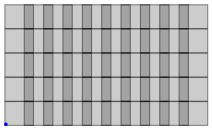
Le même tableau avec une forme de $(80, 50)$, tuilé en (10) blocs de taille $(50, 8)$, en utilisant une matrice d'ajustage de $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ et une matrice de pavage de $\left(\begin{smallmatrix} 8 \\ 0 \end{smallmatrix}\right)$.



FENÊTRE GLISSANTE. Cela représente des accès par bloc où les blocs ne tuilent pas parfaitement certaines dimensions du tableau, avec des blocs qui se chevauchent et qui provoquent certains éléments du tableau de faire partie de multiples motifs. Sur les dimensions du glissement, les vecteurs de pavage ont des valeurs non-nulles inférieures aux dimensions correspondantes du motif, des valeurs qui représentent les *pas* de la fenêtre sur cette dimension.

Exemple.

En laissant inchangées les matrices du pavage et d'ajustage de l'exemple par bloc, et par l'élargissement du motif à un bloc de taille $(10, 12)$, on se retrouve avec un accès en fenêtre glissante sur la première dimension du tableau, avec un pas de 8.



2.4 DÉFINITION STATIQUE

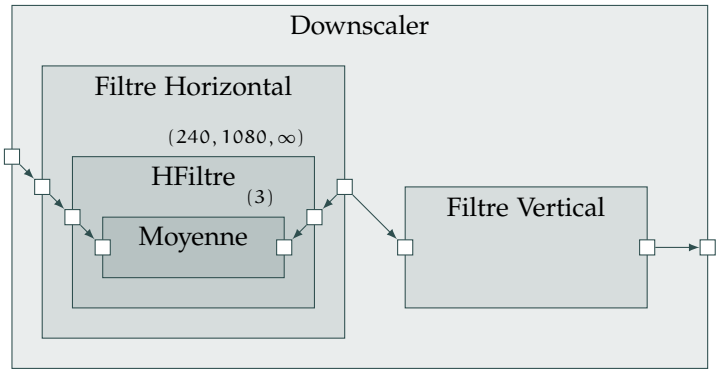
Dans les sections précédentes nous avons donné les constructions des relations de dépendance entre les éléments des tableaux pour les trois catégories de tâches : élémentaire, composée ou répétition. En utilisant la décomposition hiérarchique, par induction, la relation de dépendance peut être dérivée entre tous les appels des tâches élémentaires. Seules les tâches élémentaires peuvent avoir associées de calcul, donc seules ces tâches seront exécutes. La description ARRAY-OL est utilisé pour exprimer les accès aux données et en même temps les dépendances de données, donc les dépendances d'exécution entre les tâches élémentaires.

Chaque tâche élémentaire sera exécutée en concordance avec un espace de répétition total donné par la concaténation des espaces de répétition trouvés en parcourant tous les niveaux de hiérarchie à partir du niveau le plus haut jusqu'au niveau de la tâche élémentaire.

Exemple. La Figure 2.4 montre le calcul de cet espace sur un exemple, toujours l'application Downscaler où la tâche *HFiltre* peut être encore décomposée dans une sous-tâche répétée, qui calcule chaque pixel de sortie comme une moyenne à partir de plusieurs pixels d'entrée[10].

[10] Sur la figure seules les répétitions sont montrées.

Les spécifications suivant la définition ARRAY-OL *ne sont pas toutes valides.* Certaines ne peuvent même pas respecter l'hypothèse de base de l'assignation unique. D'autres peuvent définir des cycles dans les relations de dépendance et ainsi peuvent provoquer des blocages. Des règles faciles et vérifiables statiquement pour assurer qu'une application est ordonnançable statiquement sont définies dans [14]. Nous rappellerons ces règles par la suite.



La tâche élémentaire *Moyenne* a un espace de répétition total de $(240, 1080, \infty, 3)$ obtenu par la concaténation des tous les espaces de répétition en descendant la hiérarchie jusqu'au niveau de la tâche élémentaire. Cet espace de répétition donne un nombre d'exécutions de $240 \times 1080 \times 3 = 777600$ fois de la tâche pour chaque image.

FIG. 23: La répétition totale d'une tâche élémentaire

CONDITION DE LA DÉFINITION STATIQUE

Une application ARRAY-OL est définie statiquement et ainsi ordonnançable statiquement si la relation de dépendance entre les appels des tâches élémentaires est un ordre partiel strict. Effectivement, un quelconque ordonnancement qui respecte cet ordre partiel calcule les mêmes valeurs de sortie en utilisant les mêmes entrées.

Les règles proposées sur le langage pour assurer la définition statique sont :

A. les tailles de tous les tableaux sont des valeurs connues, et pas des paramètres ;

B. les tableaux de sortie d'une tâche sont toujours produits complètement si les tableaux d'entrée sont définis complètement ;

C. les cycles ne sont pas permis dans le graphe d'une tâche composée.

Le langage qui respecte ces règles est appelé ARRAY-OL *statique*. Pour chaque type de tâches, on peut démontrer par induction que ces règles garantissent l'existence d'un ordre partiel strict (démontré par Boulet dans [14]) :

TÂCHES ÉLÉMENTAIRES. Ces tâches produisent entièrement les tableaux de sortie à partir des entrées et comme la tâche élémentaire est la seule à s'exécuter dans ce contexte, la relation de dépendance est évidemment un ordre partiel strict.

TÂCHES COMPOSÉES. Dans une tâche composée, le graphe des sous-tâches ne permet pas de cycles et ainsi est un graphe acyclique orienté, donc il définit un ordre partiel strict entre ces nœuds.

TÂCHES RÉPÉTÉES. Pour les répétitions, il faut vérifier que tous les tableaux sont produits entièrement. Puisque la répétition est parallèle, on n'a pas de dépendances entre des appels différents de la tâche répétée. Une conséquence directe de la règle de la production entière des tableaux est qu'une répétition doit obligatoirement paver exactement les tableaux de sortie. Vérifier cette condition peut se faire facilement en utilisant des outils de calcul

polyédrique, tel que SPPoC[11] [16]. La méthodologie de vérification en utilisant le calcul polyédrique est décrite en détail dans [14]. Il faut observer que la règle de la production entière et unique des éléments est liée directement à l'assignation unique et que la spécification d'une répétition qui ne respecte pas cette règle n'est pas sémantiquement valide et dans ce cas on peut utiliser les méthodes polyédriques pour vérifier que la spécification est correcte.

[11] http://www.lifl.fr/west/sppoc/

Avec ARRAY-OL, *la spécification des répétitions est la plus compliquée à faire et des outils peuvent aider le modeleur à visualiser ou vérifier sa validité.*

Les règles pour garantir la définition statique ne sont que des règles pour assurer que la spécification corresponde aux principes d'ARRAY-OL, comme l'assignation unique et la production intégrale des tableaux de sortie. L'interdiction de cycles dans le graphe des tâches pose des restrictions dans l'ensemble d'applications qu'on peut exprimer avec ARRAY-OL, mais se sont des restrictions qui correspondent au domaine d'applications envisagé par ARRAY-OL, le traitement intensif de signal avec des sémantiques orientées flot de données.

Nous avons présenté le langage ARRAY-OL dédié à la spécification des applications de traitement intensif de signal. Il permet la modélisation du parallélisme complet disponible dans l'application par des décompositions data-parallèles et, à un haut niveau d'abstraction, des mécanismes d'expression des accès uniformes aux sous-tableaux. Le langage est d'assignation unique et manipule des tableaux multidimensionnels, en se concentrant sur l'expression compacte des accès multidimensionnels.

2.5 POUVOIR D'EXPRESSION

Essayons maintenant de faire une comparaison entre ARRAY-OL et les autres modèles de calcul présentés. On s'intéresse surtout aux langages avec lesquels ARRAY-OL partage des caractéristiques. Comme on l'a déjà dit, *la sémantique* ARRAY-OL *est celle d'un langage fonctionnel de premier ordre qui manipule des tableaux multidimensionnels, statiquement ordonnançable*. On s'intéresse surtout à l'aspect multidimensionnel et au pouvoir d'expression.

2.5.1 *Positionnement*

Des différentes approches pour la spécification des applications de traitement intensif de signal ont été proposées au fil des années. Nous avons présenté dans les sections précédentes les principes d'une partie des langages existants, correspondant aux différentes approches.

De dire que certaines propositions de langages sont meilleures que des autres risque d'être très subjective. Ce qu'on peut affirmer est que chaque approche a ces avantages et désavantages et des langages sont plus expressifs que des autres, mais la simplicité peut être vue aussi comme un avantage.

Une modélisation représente une abstraction de la réalité et essaye de capter certaines caractéristiques essentielles qui peuvent être exploitées par la suite. Dans le contexte des applications de traitement intensif de signal, une caractéristique que nous considérons essentielle est l'uniformité des accès aux structures de données multidimensionnelles.

Quand on parle d'optimisations de code, l'objectif est d'exécuter plus vite en consommant moins de ressources (taille, mémoire, énergie, etc.).

La plupart des techniques d'optimisation source à-source travaillent autour les boucles qui représentent la partie principale des calculs (même pour des applications qui ne sont pas nécessairement orientées traitement intensif) et où l'aspect répétitif de l'exécution offre des possibilités d'optimisations complexes, un domaine de recherche considérablement exploré par les techniques d'optimisation basées sur de transformations de boucles.

Dans le cas des applications de traitement intensif de signal, l'aspect régulier des calculs est encore plus pertinent et cela facilite davantage les techniques d'optimisation. Ces techniques d'optimisation sont très complexes et elles montrent plus d'efficacité dans le cas du code bien structuré[12].

[12] *Exemple :* de nids de boucles parfaits.

Le langage ARRAY-OL a été conçu pour la modélisation visuelle de l'aspect régulier des calculs dans les applications de traitement intensif de signal, par une représentation visuelle de boucles data-parallèles. ARRAY-OL exprime uniquement les vraies dépendances de données dans une vision orientée flot de données.

ARRAY-OL utilise un formalisme visuel[13] qui correspond mieux au concept de modélisation de haut niveau. Les valeurs des formes multidimensionnelles et des accès uniformes par des tilers sont spécifiées sous une forme textuelle, mais expriment la représentation formelle des espaces multidimensionnels et de vecteurs dans ces espaces, qui pour plus de trois dimensions rendent délicate une représentation purement visuelle.

[13] ARRAY-OL ne permet pas l'accès direct aux variables (tableaux), indices, etc.

Évidement, une spécification visuelle est moins expressive que l'expression textuelle qui est beaucoup plus flexible. Le langage Alpha partage beaucoup de caractéristiques avec ARRAY-OL en utilisant un langage formel textuel et des polyèdres comme structures de données. Nous allons faire une comparaison plus détaillée entre les deux formalismes dans la sous-section 2.5.5.

Pour ARRAY-OL, l'aspect statique est très important ; une définition statique permet le calcul d'un ordonnancement statique et de simplifier les mécanismes d'exécution. Les règles définis sur le langage pour garantir la définition statique contraint la spécification en ARRAY-OL à un noyau statique où toutes les valeurs des tailles des tableaux sont connues numériquement. Sur ce noyau statique, des extensions dynamiques peuvent être définies : paramètres, contrôle, etc.

Les caractéristiques du langage ARRAY-OL le reprochent plus aux langages de flot de données synchrones (Syncronous DataFlow et ses extensions) et par la suite nous allons faire une comparaison plus détaillée entre les deux approches, en regardant surtout l'aspect multidimensionnel et le pouvoir d'expression des accès uniformes.

2.5.2 *De vrais tableaux multidimensionnels*

Nous avons noté auparavant que, pour exprimer des applications multidimensionnelles, il est impératif de pouvoir manipuler des tableaux multidimensionnels ou des flots de données multidimensionnels dans le cas d'applications orientées flot de données. Il est important de bien faire la différence entre des flots de données monodimensionnels avec des éléments qui peuvent être des tableaux multidimensionnels et des « vrais » flots de données multidimensionnels. C'est le cas des modèles comme SDF et StreamIt. Malgré le fait qu'un tableau monodimensionnel avec des tableaux bidimensionnels comme éléments, par exemple,

Des constructions de base, comme l'accès par blocs, ne sont pas réalisables sur ces « faux » tableaux multidimensionnels.

peut être vu comme un tableau tridimensionnel, dans le cadre de l'approche flot de données, la multidimensionnalité de ce tableau ne peut pas être utilisée en totalité ; les motifs d'accès doivent contenir des éléments entiers, donc les tableaux qui représentent les éléments sont obligatoirement traités comme des entités indivisibles.

2.5.3 *Comparaison entre* Array-OL *et GMDSDF*

Les langages dérivés de SDF et Array-OL sont très similaires et, bien qu'ils soient différents dans leur forme, ils partagent un certain nombre de principes, comme :

- les structures de données devront avoir les dimensions multiples visibles ;
- un ordonnancement statique devrait être possible avec des ressources limitées ;
- le domaine d'applications est le même : les applications de traitement intensif de signal.

Une comparaison détaillant des modélisations avec Array-OL et GMDSDF est disponible dans [33]. Cette comparaison s'intéresse surtout à l'aspect motifs non-parallèles avec les axes et des conclusions intéressantes ont été pu tirées :

LES INFORMATIONS À DONNER. Le nombre d'informations à donner est plus grand pour Array-OL ; tous les tilers doivent être complétés, mais la difficulté de calcul des données n'est pas très élevée. En revanche en GMDSDF, même si le nombre d'informations est plus réduit, le concepteur doit être capable de comprendre le calcul de l'ordonnancement et de le vérifier ; les calculs à faire peuvent être vraiment difficiles.

CONSTRUCTION DES MOTIFS. La désignation des points à consommer ou à produire ne se fait pas de la même façon dans nos deux modèles. L'utilisation des matrices support pour GMDSDF permet de désigner des ensembles de points de forme plus irrégulière qu'en Array-OL, même des motifs qui ne sont pas des parallélépipèdes. Nous ne sommes pas sûr que la désignation de tels ensembles de points représente un avantage quelconque pour GMDSDF. En effet, il est tout à fait possible de prendre une boîte englobante pour effectuer la modélisation en Array-OL et surtout il ne nous est jamais arrivé de rencontrer des tâches ayant besoin d'une manipulation de données semblable. De plus en Array-OL, contrairement à GMDSDF il est possible de désigner des ensembles disjoints de points.

TABLEAUX TORIQUES. Il s'agit d'une possibilité réellement intéressante, par exemple des applications où une des dimensions toriques représente les sonars se trouvant autour d'un sous-marin et où une autre représente une dimension fréquentielle. L'aspect torique est une prémisse de base d'Array-OL, différemment de tous les langages basés sur SDF.

GARDER DES INFORMATIONS D'ÉTAT. GMDSDF permet d'utiliser les « états » et les « délais » ce qui n'était pas le cas d'Array-OL[14]. Pourtant ils sont indispensables à l'établissement de calculs simples comme la somme de vecteurs.

NOMBRE DE DIMENSIONS. La différence la plus notable entre GMDSDF et Array-OL est sans doute que certaines des démonstrations

[14] Problème résolu avec l'extension d'Array-OL avec des dépendances uniformes proposée dans cette thèse et présentées en section 2.6.

servant de support aux principes de GMDSDF ne sont valides que pour des applications ayant au plus deux dimensions. ARRAY-OL fonctionne avec des applications multidimensionnelles sans aucune restriction sur le nombre maximum de ces dimensions. MDSDF, lui, n'est pas limité du point de vue des dimensions, en revanche il ne permet qu'une manipulation rudimentaire des données.

Nous avons vu que le modèle GMDSDF a certains avantages en comparaison avec ARRAY-OL, mais surtout des désavantages causés par la complexité des calculs pour des applications avec plus de deux dimensions.

2.5.4 *Désavantages de l'approche flots de donnés synchrones*

En regardant les langages de la famille *flot de données synchrone*, on peut observer que, en partant de SDF, des fonctionnalités ont été ajouté pour étendre de plus en plus le langage.

En suivant l'évolution SDF, MDSDF, GMDSDF, WSDF, on voit bien l'introduction successive des concepts pour pouvoir exprimer la multi-dimensionnalité, les motifs non-parallèles avec les axes et ensuite les fenêtres glissantes. Toutes ces extensions sont faites de telle manière qu'elles restent compatibles avec le modèle antérieur. Rendre toutes ces extensions compatibles permet la coexistence des acteurs décrits en différents langages dans la même application. Néanmoins, nous considérons que ce choix a aussi des effets négatifs sur le pouvoir d'expression, comme on va voir par la suite.

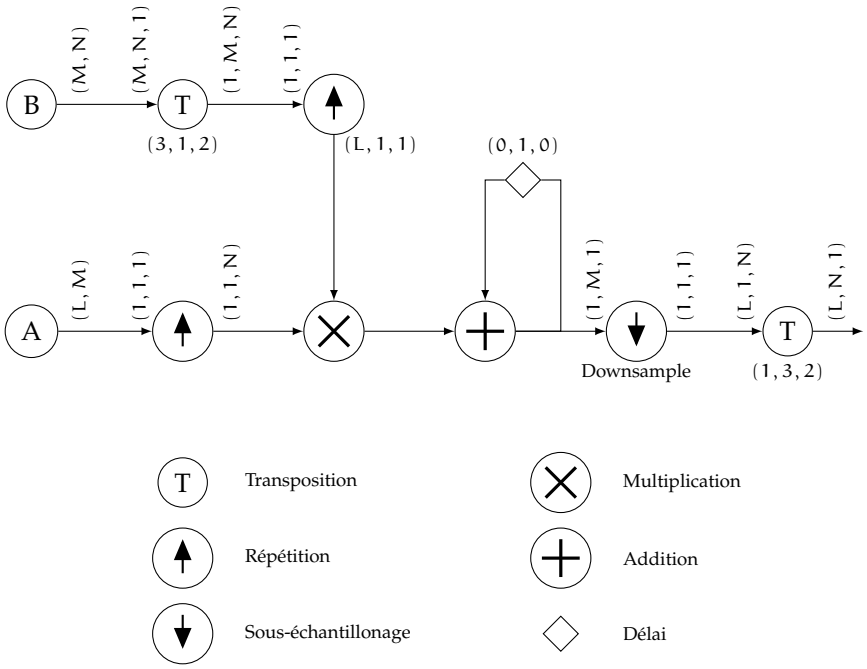
Nous allons nous limiter aux motifs parallèles avec les axes et on va voir que même MDSDF, qui devrait être capable d'exprimer ces accès, peut avoir des difficultés, causées par le fait d'avoir gardé des principes de SDF qui ne sont parfaitement adaptées aux flots multidimensionnels, à notre avis.

Flot de motifs, et non pas des motifs tuilés dans des tableaux

Dans l'approche flots monodimensionnels de SDF, l'utilisation des FIFOS pour les motifs d'entrées et de sortie est évidente. Le flot de données a la forme d'un flot infini de motifs. En passant à un modèle multidimensionnel, dans MDSDF, ces principes ont été gardés, en utilisant des structures FIFOS multidimensionnelles infinies sur chaque dimension. Pour rester fidèles aux principes SDF et pour simplifier le calcul de l'ordonnancement statique, chaque dimension est vue comme un flot monodimensionnel équivalent à la représentation SDF.

Plusieurs inconvénients viennent de ces choix, visibles sur un exemple d'une multiplication des deux matrices, repris de [69] et montré sur la Figure 24 :

DIMENSIONS BORNÉES. Les flots de données n'ont pas de dimensions bornées, donc toutes les dimensions des tableaux peuvent être infinies. La définition des tableaux bornés peut être simulée par l'introduction des acteurs qui produisent ces tableaux entièrement, comme dans l'exemple de la Figure 24. Cela peut contraindre à garder les deux matrices en mémoire, contrairement à une fonctionnalité de type pipeline, où seules des motifs des matrices sont placés dans la mémoire.



Les deux matrices opérandes sont projetés dans l'espace 3D des indices, en utilisant des acteurs clés de *transposition*, *répétition* ou *sous-échantillonnage*. Les opérations de multiplication et d'addition se font élément par élément. L'ensemble *addition* et *délai* représente une construction d'état, nécessaire pour décrire l'opération de somme d'un tableau.

FIG. 24: Multiplication des matrices en MDSDF

MIXER LA DIMENSIONNALITÉ. Le changement du nombre des dimensions dans une application se fait difficilement et seulement à l'aide des acteurs clés ; sur la Figure 24, l'acteur *Répétition* va répéter des éléments sur une dimension ou l'acteur *Transposition* qui va interchanger des dimensions entre elles, etc. D'après nos connaissances, des constructions habituelles comme la linéarisation des dimensions ne sont pas exprimables en MDSDF.

CONSOMMATION MULTIPLE. En préférant l'approche flots des motifs aux motifs pavés dans des tableaux, les langages flot de données synchrone interdisent les consommations multiples de données. Ce n'est pas la consommation d'un flot par plusieurs acteurs, qui est possible en utilisant des nœuds de branchement, *fork*, mais des consommations des mêmes éléments par plusieurs invocations d'un acteur, le cas des fenêtres glissantes discutées dans la section suivante.

La Figure 25 illustre la modélisation de la même application de multiplication des matrices en ARRAY-OL. En regardant cette modélisation, on peut retrouver facilement les tailles des tableaux et des motifs, ainsi que les espaces de répétitions. C'est la première étape dans la modélisation ARRAY-OL, avec la construction des graphes de dépendances.

Des outils visuels pour aider dans la modélisation sont disponibles à `http://www.lifl.fr/west/aoltools/`.

L'étape de la spécification des répétitions, qui implique la saisie des tilers, est la plus compliquée pour le concepteur, mais a l'avantage qu'avec les concepts de tiler on peut modéliser la plupart des constructions des motifs uniformément espacés : pavage par bloc, fenêtres glissantes, accès cyclique, motifs non-parallèles avec les axes, etc. Une

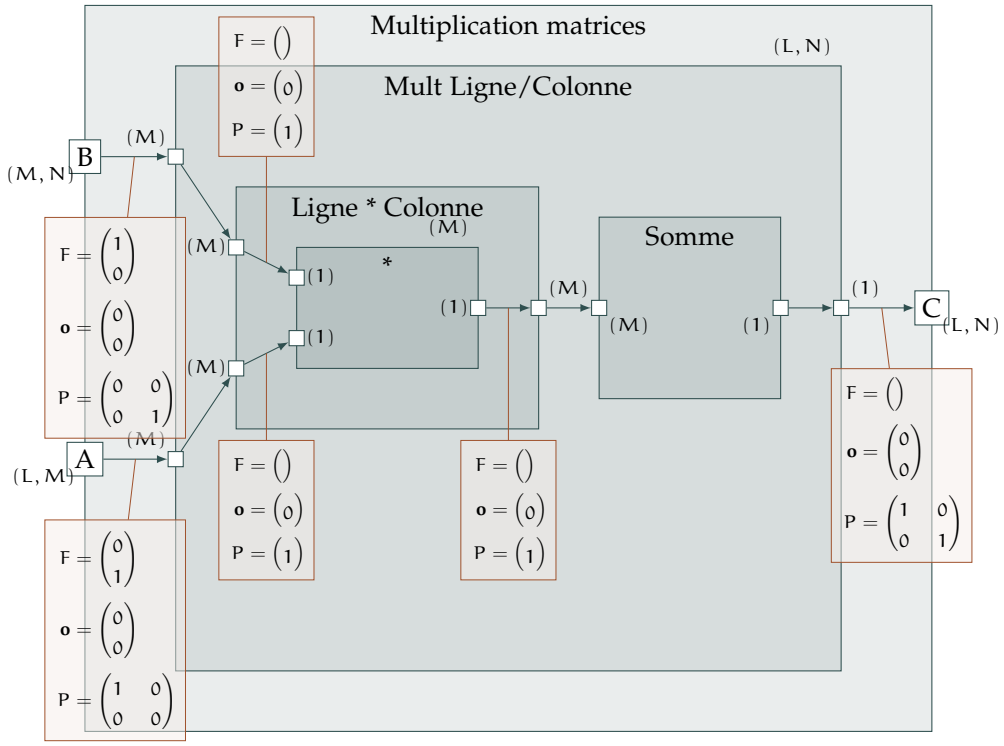


FIG. 25: Multiplication de matrices en ARRAY-OL

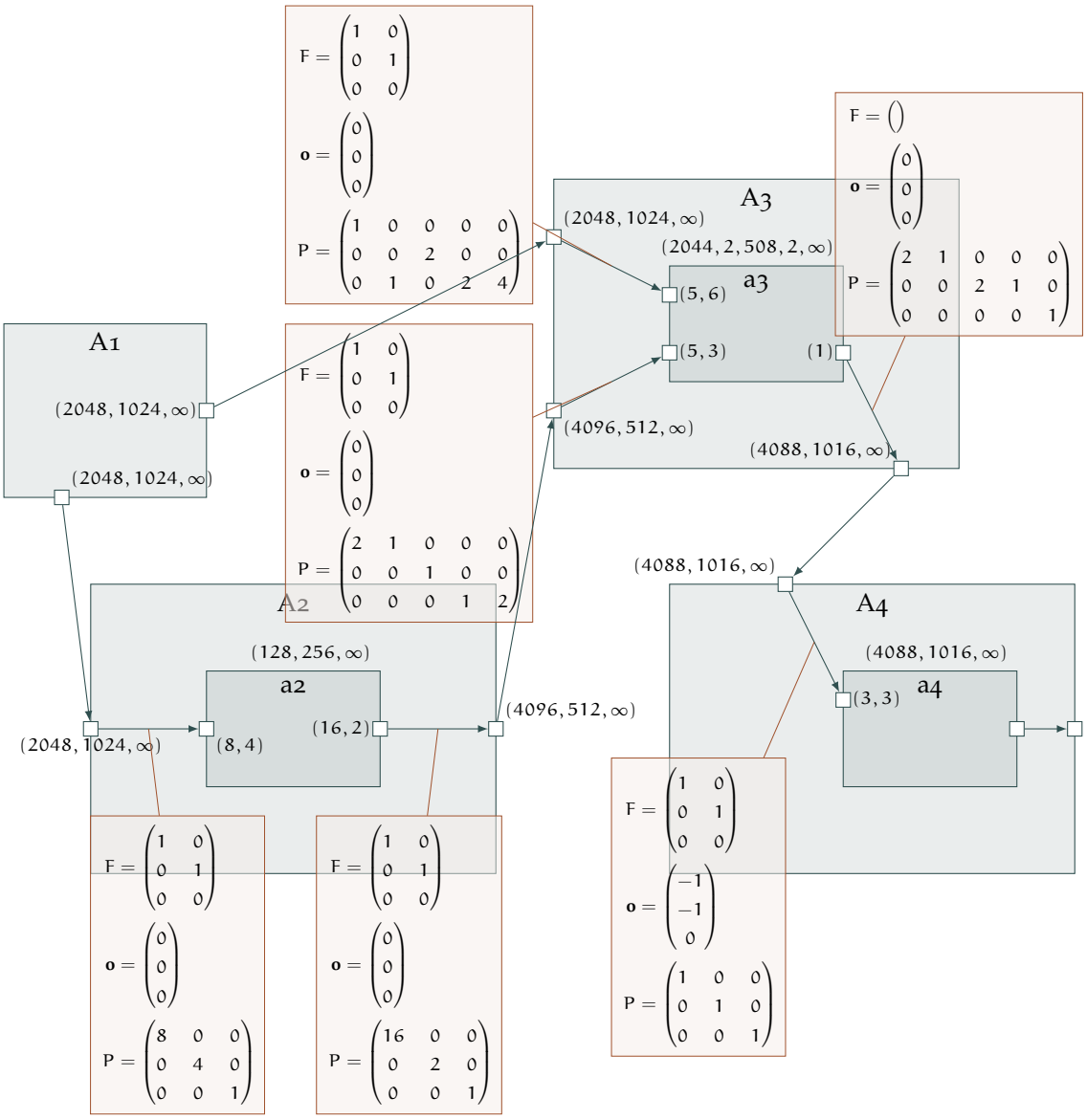
fois le concept de *tiler* maîtrisé, la modélisation en ARRAY-OL devient facile.

La multiplication des matrices en ARRAY-OL, la Figure 25, montre aussi les limites du modèle ARRAY-OL : on ne peut pas exprimer des constructions d'état nécessaires pour modéliser sous la forme répétitive le calcul de la somme d'un tableau. En modélisant cette opération comme une tâche élémentaire on perd le parallélisme disponible et cela peut influencer la qualité de la modélisation.

Accès en fenêtre glissante

Nous avons vu dans l'étude des langages orientés flot de données que l'approche prédominante est évolutive ; à partir d'un langage existant, des fonctionnalités additionnelles sont ajoutées en utilisant des nouveaux concepts, en restant toujours compatible avec le langage d'origine. Le modèle WSDF est l'illustration parfaite de cette approche. Dans la Figure 9, sous-section 1.6.4, nous avons montré une application contenant des accès de type *fenêtre glissante* modélisés avec WSDF. Plusieurs concept nouveaux ont été ajouté dans le but de modéliser spécialement ces algorithmes. On voit bien que ces concepts alourdissent beaucoup la modélisation et en plus ils servent uniquement à la modélisation des fenêtres glissantes. La Figure 26 montre la même application modélisée avec ARRAY-OL.

Le modèle ARRAY-OL est capable d'exprimer facilement des algorithmes de fenêtre glissantes ; ces algorithmes sont traités comme tout autre accès multidimensionnel. Il faut remarquer que la Figure 26 exprime exactement les mêmes accès que celles de la Figure 9, même si la



Le tableau de sortie de la tâche *A3* a une taille plus petite que dans la modélisation en WSDF ; en Array-OL on ne dispose pas d'un concept pour exprimer que plusieurs images sont traité en continue par l'accès en *fenêtre glissante*. La vision Array-OL est que si plusieurs images sont traité en continue, la modélisation doit la refléter. En mettant l'origine a $(-1, -1, 0)$, le dépassement des frontières se fait aussi sur la première ligne et la première colonne, pour avoir une modélisation conforme au modèle en WSDF.

Fig. 26: Fenêtres glissantes en Array-OL

nécessité d'avoir des *jetons virtuels* et des *unions des jetons virtuels* nous semble inadéquate[15].

[15] C'est pourquoi sur la Figure 26 on arrive à avoir une répétition à 5 dimensions pour des tableaux 3-dimensionnels.

Observation. Pour exprimer que, dans un flot d'images, quatre images successives sont traitées comme une seule image, en Array-OL, on transforme la forme du flot dans un autre flot ou l'image est quatre fois plus grande et contient les quatre images successives.

La motivation pour ce choix est de pouvoir exprimer des accès continus entre des images successives. En Array-OL, pour exprimer des accès similaires, la solution est de transformer la forme des tableaux,

en utilisant la construction de *Reshape*, présenté ultérieurement dans la section 3.4.

LE TRAITEMENT DES FRONTIÈRES. L'expression du traitement des frontières a été ajouté en WSDF. La question qui se posé est la suivante : que se passe-t-il quand les fenêtres d'accès sont à la frontière des tableaux ? Il y a plusieurs possibilités :

1. l'accès ne doit pas dépasser les frontières, cas où l'image de sortie aura une taille réduite en comparaison avec celle d'entrée ;
2. l'accès dépasse les frontières, en utilisant une approche cyclique ;
3. les frontières peuvent être entendues avec des éléments par défaut.

Le premier cas est l'approche habituelle, le désavantage étant la réduction des taille des tableaux. Le troisième cas a été choisi comme solution en WSDF, l'accès cyclique n'étant pas conforme à l'approche flot de données. En ARRAY-OL, l'accès cyclique est une prémisse de base du langage et donc cette approche est naturelle. Pour pouvoir exprimer des extensions des frontières on peux faire appel a un concept nouveau, le *lien par default*, utilisé pour exprimer des dépendances uniformes en ARRAY-OL, présenté par la suite dans le section 2.6.

Dans la modélisation des algorithmes avec des accès en fenêtre glissante, ARRAY-OL est capable d'exprimer ces constructions sans grands efforts. Le modèle WSDF est conçu exclusivement pour la spécifications de ces algorithmes et dispose de concepts spécialisés pour chaque caractéristique nécessaire. Même avec tous les nouveaux concepts ajoutées, WSDF ne peut toujours pas modéliser des accès cycliques. ARRAY-OL ne peut pas exprimer des accès en continu sur des éléments qui ne sont pas sur la même dimension, mais dispose de mécanismes de « restructuration » des tableaux, par le changement de leurs formes en utilisant le concept de *Reshape*, présenté dans la section 3.4. L'extension des frontières est aussi possible en utilisant le concept de *lien par défaut*.

Une modélisation plus fidèle à l'application décrite en WSDF, en utilisant certains concepts additionnels introduits par la suite en ARRAY-OL, *est disponible dans Appendice A.*

2.5.5 ARRAY-OL *et Alpha*

Le langage Alpha, présenté dans la sous-section 1.4.1, partage de propriétés importantes avec ARRAY-OL : les deux sont des langages fonctionnels orientés flot de données et les deux langages expriment les dépendances de données dans une application en utilisant un formalisme d'assignation unique.

Mais, en utilisant un formalisme textuel et une représentation des algorithmes sous forme d'équations récurrentes, le langage Alpha permet la spécification des applications plus complexes, là où le formalisme visuel d'ARRAY-OL permet l'expression des accès uniformes qui peuvent être décrits avec le concept de tiler : les coordonnées des éléments des motifs dans le tableau sont des combinaisons linéaires des indices des répétitions et des coordonnées des éléments dans les motifs.

Les structures de données manipulées sont différentes, des tableaux multidimensionnels cycliques pour ARRAY-OL et des polyèdres pour Alpha. Une structure polyédrique est plus générale que des tableaux multidimensionnels où des espaces avec des formes plus complexes ne sont pas exprimables[16]. La restriction est toujours causée par le formalisme visuel qui ne permet pas l'utilisation des variables ou l'identification directe des dimensions des tableaux.

[16] *Exemple :* Un espace triangulaire.

Observation. Le noyau statique ARRAY-OL ne permet pas de définir des structures de données qui ne sont pas rectangulaires. Cela n'exclut pas l'introduction des extensions : une possibilité que nous avons enquêté est le paramétrage des dimensions des tableaux et la spécification des contraintes entre ces dimensions. Pratiquement, cela se traduit par une représentation polyédrique composée d'un espace multidimensionnel contraint par d'hyperplans.

Néanmoins, nous considérons que l'utilisation des tableaux multidimensionnels correspond largement au domaine d'applications qu'ARRAY-OL cible :

- la plupart des structures de données sont des tableaux multidimensionnels ou ils peuvent être englobés dans une forme multidimensionnelle ;
- en ARRAY-OL, même si les tableaux et les motifs ont une forme rectangulaire, les placements des motifs dans les tableaux (ce que nous appelons tuiles) peuvent prendre des formes complexes non rectangulaires ;
- les structures de données multidimensionnelles qu'ARRAY-OL utilise au niveau spécification peuvent être traduites directement dans des structures de données équivalentes dans le code généré ;
- l'aspect cyclique natif en ARRAY-OL n'est pas géré par Alpha.

Même si les deux langages (ARRAY-OL et Alpha) partagent des principes en commun, le choix visuel/textuel de la spécification apporte des différences importantes dans l'expressivité et la forme. Un langage textuel est plus flexible en termes d'expressivité de ses constructions, mais n'apporte pas l'aspect visuel lié au concept de modélisation de haut niveau.

Nous avons vu dans la comparaison avec les langages de spécification synchrones que la décomposition répétitive d'ARRAY-OL permet l'expression des accès par motifs uniformes usuels, mais pose des problèmes dans l'expression des cycles dans la spécification, nécessaires pour exprimer de structures d'état. Pour surmonter cette limitation, nous proposons une extension du langage qui permet l'expression des dépendances uniformes entre les répétitions data-parallèles.

2.6 MODÉLISATION DES DÉPENDANCES UNIFORMES

Afin d'exprimer les informations nécessaires pour une spécification complète des algorithmes orientés flot de données, les langages comme ARRAY-OL se basent sur des formalismes spéciaux. Un des aspects rencontrés couramment dans ces langages est l'approche orientée composants dans laquelle une application est successivement décomposée en tâches qui sont connectées et communiquent à travers des tableaux. Pour maintenir les sémantiques de flot de données, les cycles sont interdits dans les graphes des connexions entre les tâches. Combiné avec la contrainte d'assignation unique, cela interdit aux tâches de conserver les informations d'état.

Dans ce contexte, le concept de *délai* disponible dans SDF et ensuite dans la plupart de ses extensions revêt une grande importance pour un langage de flot de données, en permettant la construction des cycles et des tâches avec des boucles sur elles-mêmes, ce qu'on appelle des constructions d'*état*. Une telle construction permet un acteur d'utiliser des données qu'il a produit mais dans une exécution antérieure.

Un langage « complet » orienté flot de données doit impérativement être capable d'exprimer ce genre de constructions. ARRAY-OL étant un langage plus expressif que les langages de la famille *flot de données synchrone*, les concepts capables d'exprimer des cycles dans le graphe doivent être par conséquent plus complexes.

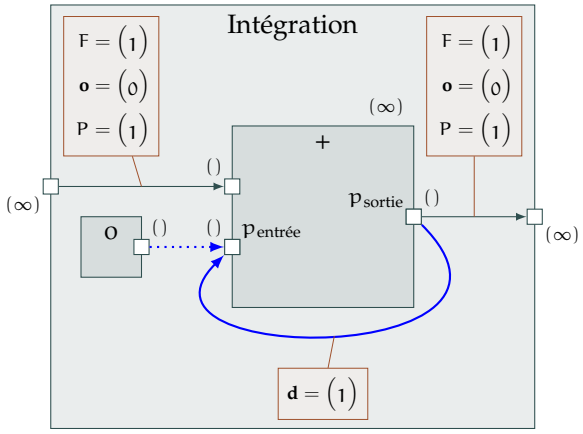
Dans SDF, un délai est représenté par des jetons initiaux disponibles sur un lien. Avec l'extension multidimensionnelle de MDSDF, les délais sont devenus aussi des jetons multidimensionnels, représentés par des lignes et des colonnes initiales dans le cas bidimensionnel, ou des hyperplans pour plus de dimensions. Ces valeurs initiales disponibles sur un lien impliquent que la production de jetons sur le lien est déplacée avec les valeurs correspondantes, alors que la consommation ne l'est pas. Des données déjà disponibles sur le lien permettent à un acteur d'avoir une boucle sur lui-même et de consommer des jetons produits par lui-même dans une itération antérieure, sans bloquer l'exécution, constructions appelées *états*.

En ARRAY-OL, *il n'y a pas d'ordre prédéfini d'exécution et ainsi le concept de jetons initiaux n'est pas adéquate.*

Plutôt que d'exprimer des jetons initiaux disponibles sur une connexion, nous avons choisi d'exprimer des dépendances uniformes entre les répétitions de la même tâche répétée, ce que nous appelons des *dépendances interrépétition*.

2.6.1 *Exemple introductif*

Pour être en mesure de représenter des boucles contenant des dépendances interrépétition, nous avons ajouté la possibilité de modéliser des dépendances uniformes entre les tuiles produites et celles consommées par une composante répétée. La Figure 27 contient une représentation graphique de cette construction.



La dépendance est représentée par la boucle autour la tâche répétée +. Le connecteur en pointillé représente le lien par défaut.

FIG. 27: Une simple dépendance interrépétition

Formellement, une dépendance interrépétition connecte un port de sortie, p_{sortie} sur la Figure 27, d'une tâche répétée avec un de ces ports d'entrée, $p_{entrée}$ sur la même figure. *La forme des deux ports connectés doit impérativement être identique.* Le connecteur de dépendance a associé un

vecteur de dépendance $\mathbf{d}$ qui définie la *distance de dépendance* entre les répétitions dépendantes,

$$\mathbf{r}_{\text{dép}} = \mathbf{r} - \mathbf{d}. \tag{2.4}$$

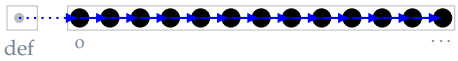
Cette dépendance est uniforme, ce qui signifie identique pour toutes les répétitions. Quand la source de la dépendance est en dehors de l'espace de répétition, une valeur par défaut est utilisée, définie par le *lien par défaut* connecté au même port d'entrée, $p_{\text{entrée}}$:

A. Quand on affirme qu'une répétition $\mathbf{r}$ dépend d'une autre $\mathbf{r}_{\text{dép}}$, cela signifie qu'au moment de l'exécution la répétition $\mathbf{r}$ prendra comme valeurs d'entrées sur le port $p_{\text{entrée}}$ des valeurs produites par la répétition $\mathbf{r}_{\text{dép}}$ sur son port de sortie p_{sortie}.

B. Quand une répétition prend des valeurs d'un lien par défaut, cela signifie que les valeurs consommées sur le port $p_{\text{entrée}}$ provient d'un port connecté avec un lien par défaut à ce port $p_{\text{entrée}}$ (le tableau entier associé à ce port si les deux ont la même forme ou une tuile de ce tableau, cas où le connecteur par défaut doit avoir un tiler associé, comme nous allons voir en plus de détails par la suite).

Dans l'exemple d'une simple intégration discrète de la Figure 27, les motifs (et aussi les tuiles) sont des points singuliers. Le vecteur de dépendance $\mathbf{d} = (1)$ précise que chaque répétition $\mathbf{r}$ dépend de la répétition $\mathbf{r}_{\text{dép}} = \mathbf{r} - \mathbf{d} = \mathbf{r} - (1)$. La représentation graphique des dépendances dans l'espace de répétition est illustrée sur la Figure 28 Dans ce cas, la dépendance interrépétition est utilisée pour exprimer que la valeur de sortie d'une répétition est utilisée comme entrée par la prochaine répétition.

La dépendance interrépétition introduit un ordre dans l'espace d'une répétition et c'est pourquoi on peut parler de répétition précédente/suivante.



Dans l'espace de répétition monodimensionnel, la dépendance interrépétition $\mathbf{d} = (1)$ exprime que chaque répétition dépend de celle qui la précède. La première répétition dépend d'une qui est en dehors de l'espace de répétition et ainsi elle va prendre la valeur par défaut comme entrée.

FIG. 28: Les dépendances dans l'espace de répétition de l'*Intégration*

À l'exécution, chaque répétition prendra comme entrées deux valeurs sur ses deux ports d'entrée, une valeur d'entrée d'une tuile et le résultat de la répétition précédente. Ces valeurs sont additionnées et le résultat est fourni sur le port de sortie, qui va en même temps servir comme entrée pour la répétition suivante et être rangée dans le tableau de sortie. Pour commencer le calcul, une valeur par défaut de 0 est pris par la répétition $\mathbf{r} = 0$, comme indiqué par le lien par défaut. Les calculs sont séquentiellisés par la dépendance interrépétition. La répétition mise à plat est montré sur la Figure 29.

La dépendance interrépétition présentée est un exemple relativement simple utilisé pour illustrer les concepts de dépendances uniformes et de lien par défaut. Le langage ARRAY-OL étendu par ces concepts permet la construction des structures plus complexes, comme des dépendances interrépétition multiples ou des dépendances non contiguës. Ces dépendances peuvent être également connectées à travers la hiérarchie pour exprimer des dépendances entre des répétitions aux différents niveaux de la hiérarchie.

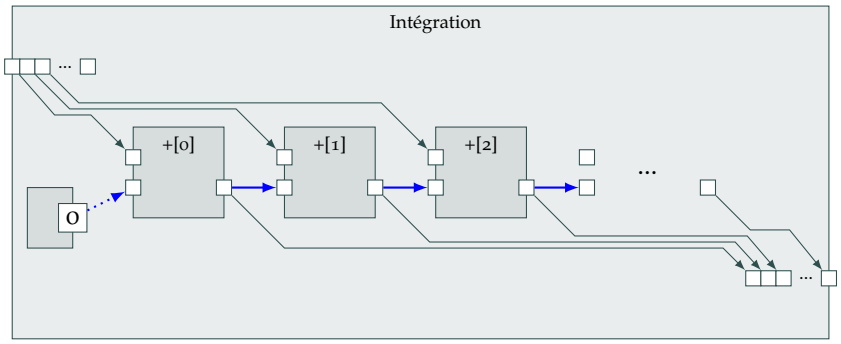


FIG. 29: Intégration : répétition mise à plat

2.6.2 *Définition de la dépendance interrépétition*

Une dépendance interrépétition complète, dans le contexte d'ARRAY-OL, qui permet la construction des dépendances complexes connectées à travers les niveaux de la hiérarchie, contient les deux éléments que nous avons vu sur l'exemple précédent :

- la dépendance uniforme étiquetée avec un vecteur de dépendance ;
- le lien par défaut ;

auxquels on ajoute deux observations :

- premièrement, le lien par défaut peut avoir comme source un port du composant qui contient la dépendance interrépétition et la tâche répétée. De cette façon, on peut établir des connexions entre des niveaux successifs de la hiérarchie ;
- deuxièmement, le lien par défaut peut connecter des ports avec des formes différentes, cas où un tiler associé au lien est nécessaire pour exprimer les relations exactes élément à élément.

Le tiler sur le lien par défaut permet d'avoir de multiples liens par défaut connectés à une dépendance interrépétition – un seul lien doit être valide pour chaque répétition qui a besoin d'une valeur par défaut.

Exemple. Un exemple montrant l'intérêt de cette construction est un espace de répétition bidimensionnel avec des valeurs différentes par défaut en fonction de la direction dans laquelle on sort de l'espace de répétition (nord, sud, est ou ouest).

Définition 1 (dépendance interrépétition)**.** La spécification formelle d'une dépendance interrépétition complète consiste en :

- un composant répété c avec un espace de répétition $\mathbf{s}_{\text{répétition}}$;
- une dépendance interrépétition dép avec un vecteur de dépendance $\mathbf{d}$; dép connecte un port de sortie p_{sortie} à un port d'entrée $p_{\text{entrée}}$ (p_{sortie} et $p_{\text{entrée}}$ doivent avoir la même forme $\mathbf{s}$ et les deux appartient à c) ;
- un ensemble de n liens par défaut déf_i ($0 \leqslant i < n$), connectant des ports de sortie p_i ($0 \leqslant i < n$) d'autres composants au port $p_{\text{entrée}}$;
- chaque lien par défaut déf_i a associé un tiler θ_i, avec l'exception du dernier qui peut manquer le tiler (cas où $p_{n\text{-}1}$ doit avoir la même forme que $p_{\text{entrée}}$) ; t représente le nombre des liens par défaut ayant des tilers associés ($n - 1 \leqslant t \leqslant n$).

En calculant les dépendances, nous retrouvons :

$$\forall\, \mathbf{r}, \mathbf{0} \leqslant \mathbf{r} < \mathbf{s}_{\text{répétition}}, \mathbf{r}_{\text{dép}} = \mathbf{r} - \mathbf{d} \tag{2.5}$$

et si la répétition de dépendance est à l'intérieur de l'espace de répétition ($\mathbf{0} \leqslant \mathbf{r}_{\text{dép}} < \mathbf{s}_{\text{répétition}}$) la répétition $\mathbf{r}$ dépend de $\mathbf{r}_{\text{dép}}$ (les valeurs produites par la répétition $\mathbf{r}_{\text{dép}}$ sur le port p_{sortie} sont consommées par la répétition $\mathbf{r}$ sur le port $p_{\text{entrée}}$) ; autrement la répétition $\mathbf{r}$ prend les valeurs d'entrées d'un des liens par défaut connectés au $p_{\text{entrée}}$.

Dans le cas où plusieurs liens par défaut sont connectés au port $p_{\text{entrée}}$, une sélection entre ces liens disponibles doit être faite. La sélection se fait en fonction de l'indice de répétition $\mathbf{r}$ et les tilers associés aux liens par défaut. En prenant chaque tiler θ_i, un élément de référence est calculé à l'intérieur du tableau qui corresponde au port p_i dans la même manière que pour un tiler normal (sous-section 2.3.3, Équation 2.2), mais sans l'usage du modulo :

$$\forall i, 0 \leqslant i < t, \mathbf{réf}_i = \mathbf{o}_i + P_i \cdot \mathbf{r}, \tag{2.6}$$

où $\mathbf{o}_i$ et P_i sont l'origine et la matrice de pavage du tiler θ_i.

PROPRIÉTÉ DE VALIDITÉ. La spécification des tilers associés aux liens par défaut doit respecter la contrainte que, pour toutes les répétitions qui ont besoin des valeurs d'entrée des liens par défaut, maximum une des références calculées avec l'Équation 2.6, $\mathbf{réf}_i$ ($0 \leqslant i < t$), est à l'intérieur du tableau qui corresponde au port p_i. Cette référence valide, $\mathbf{réf}_\nu$, vérifie donc que $\mathbf{0} \leqslant \mathbf{réf}_\nu < \mathbf{s}_\nu$, où $\mathbf{s}_\nu$ est la forme du port p_ν. Cette référence ainsi que le tiler correspondant T_ν sera utilisée pour calculer la tuile à passer au port $p_{\text{entrée}}$ de la répétition $\mathbf{r}$ comme un ensemble d'indices $\mathbf{e_i}$ vérifiant

$$\forall \mathbf{i}, \mathbf{0} \leqslant \mathbf{i} < \mathbf{s}, \mathbf{e_i} = \mathbf{réf}_\nu + F_\nu \cdot \mathbf{i} \mod \mathbf{s}_\nu \tag{2.7}$$

où $\mathbf{s}$ est la forme du port $p_{\text{entrée}}$ et $\mathbf{s}_\nu$ est la forme du port p_ν.

Si aucune des références calculées n'est valide, le connecteur par défaut sans tiler associé va être choisi. L'exclusion des tilers peut être facilement vérifiée en utilisant l'algèbre polyédrique.

EXCLUSION DES TILERS. L'exclusion des tilers peut se faire par la construction de l'ensemble de points de l'espace de répétition, $\mathbf{r}$, qui prennent des valeurs par défaut et qui donnent deux (au moins) références valides, par la vérification du système d'(in)équations :

$$\begin{cases} \mathbf{0} \leqslant \mathbf{r} < \mathbf{s}_{\text{répétition}} \\ \mathbf{r}_{\text{dép}} = \mathbf{r} - \mathbf{d} \\ \mathbf{r}_{\text{dép}} < \mathbf{0} \text{ ou } \mathbf{r}_{\text{dép}} \geqslant \mathbf{s}_{\text{répétition}} \\ 0 \leqslant i < t \\ \mathbf{réf}_i = \mathbf{o}_i + P_i \cdot \mathbf{r} \\ \mathbf{0} \leqslant \mathbf{réf}_i < \mathbf{s}_i \\ 0 \leqslant j < t \\ \mathbf{réf}_j = \mathbf{o}_j + P_j \cdot \mathbf{r} \\ \mathbf{0} \leqslant \mathbf{réf}_j < \mathbf{s}_j \\ i \neq j \end{cases} . \tag{2.8}$$

Si cet ensemble est vide alors les tilers des liens par défaut s'excluent entre eux. La vérification peut se faire en construisant l'ensemble de points par l'utilisation de Polylib[17] (qui est inclus dans SPPoC) et tester

[17] http://icps.u-strasbg.fr/polylib/

si l'ensemble est vide par la recherche d'un élément dans l'ensemble de points avec un appel vers le solveur PIP[18] [35], aussi inclus dans SPPoC. Ces opérations sont possibles parce que, comme les dimensions des tableaux sont des valeurs numériques connues, le système d'inéquations est équivalent à un système d'équations affines définissant des unions de treillis linéaires bornées que SPPoC, Polylib et PIP manipulent.

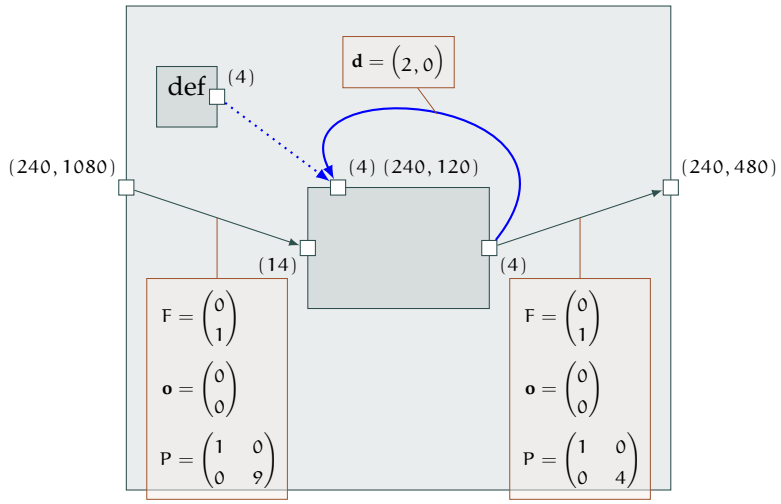
[18] http://www.piplib.org/

La construction des tilers associés aux liens par défaut pour respecter la propriété de validité (une seule référence valide pour chaque répétition) pourrait paraître complexe, mais, dans des applications réelles, des constructions avec plusieurs liens par défaut sont très rares. La plupart des exclusions entre les liens par défaut sont faites en utilisant des tilers très similaires, avec les mêmes matrices de pavage et d'ajustage et des origines déplacées. Des exemple et plus de détails sur ces constructions sont disponibles dans les sections suivantes.

Par la suite on va argumenter sur la nécessité de pouvoir exprimer des dépendances à travers les niveaux de la hiérarchie et comment la définition des dépendances interrépétition sont capables de les exprimer.

2.6.3 *Dépendances dans un espace multidimensionnel*

On commence par montrer un exemple d'une dépendance interrépétition sur un espace bidimensionnel, illustré sur la Figure 30 et qui spécifie les dépendances explicitées de la Figure 31.



Un sous-ensemble de l'application Downscaler, le filtre vertical sur une seule image. Une dépendance interrépétition a été ajoutée sur la première dimension de la répétition avec un pas de 2. Le lien par défaut fournit la valeur par défaut ; pour chaque répétition qui a une dépendance qui sort de l'espace de répétition, une « valeur » par défaut est en vérité un tableau des valeurs par défaut qui a la même forme que le port d'entrée, dans ce cas un tableau de (4) éléments.

FIG. 30: Dépendances dans un espace bidimensionnel

Dans l'espace de répétition de la Figure 31, les répétitions d'une colonne dépendent des répétitions correspondantes (sur la même ligne) de la colonne à deux pas avant dans l'ordre des indices. Les deux premières colonnes dépendent des répétitions en dehors de l'espace

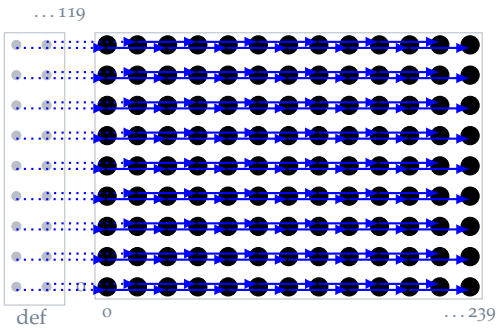
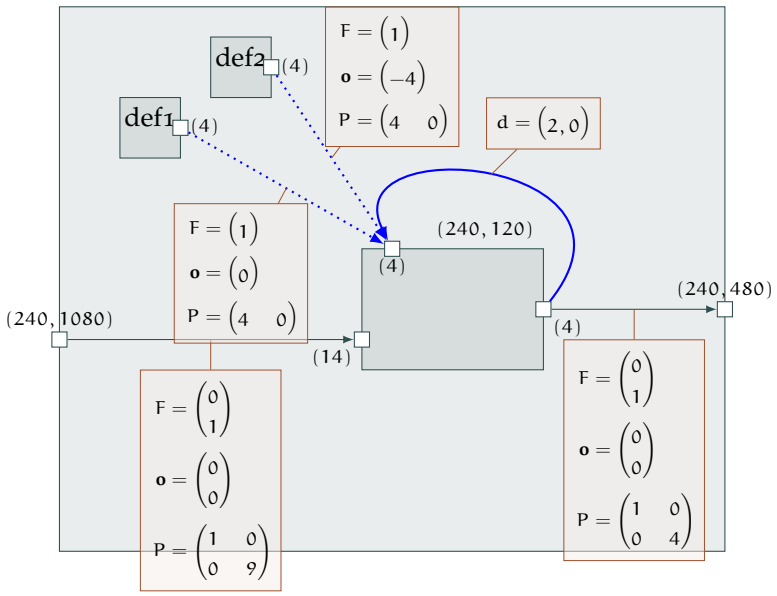


FIG. 31: Les dépendances dans l'espace de répétition 2D

de répétition et donc ces répétitions prennent des valeurs par défaut comme entrées, dans ce cas la même valeur fournie par la tâche *def*.

MULTIPLES VALEURS PAR DÉFAUT. En utilisant des liens par défaut, différentes valeurs par défaut peuvent être passées aux différentes répétitions avec des dépendances en dehors de l'espace de répétition. La Figure 32 montre l'exemple précédent avec des valeurs par défaut différentes pour chaque colonne.



Des entrées par défaut différentes sont utilisées pour les deux premières colonnes qui ont des dépendances qui sortent de l'espace de répétition. Les deux entrées sont produites par les deux tâches, *def1* et *def2*, et les tilers associés aux liens par défaut expriment les deux sous-espaces disjonctes d'application des deux entrées. La matrice de pavage de $(4\ 0)$ pour les deux tilers fait qu'un seul indice de la dimension horizontale de l'espace de répétition donne une référence dans le tableau de taille (4) et donc une seule colonne. Le choix de la colonne et fait par la valeur de l'origine, (0) pour la première colonne et (-4) pour la deuxième ($\mathbf{réf} = \mathbf{o} + P \cdot \mathbf{r} = (-4) + (4\ 0) \cdot \binom{1}{?} = (0)$, pour tous éléments de la deuxième colonne).

FIG. 32: Multiples liens par défaut

IMPACT SUR LE PARALLÉLISME DE DONNÉES. En ARRAY-OL, une tâche répétée est considérée parallèle par construction, donc ces répétitions peuvent être ordonnancées dans un ordre arbitraire par le compilateur. Une exécution parallèle des toutes les répétitions peut être considérée comme l'idéal en temps de latence, mais le choix dépend de beaucoup plus de paramètres[19]. Dans la vision ARRAY-OL, ces choix d'exécution doivent rester à la charge du compilateur/ordonnanceur et la spécification doit seulement exprimer le maximum de parallélisme.

[19] Ressources, localité des données, etc.

Sur la Figure 28, la dépendance interrépétition élimine complètement le parallélisme disponible sur la répétition, causée par la dépendance entre toutes les instances de la tâche répétée. Dans ce cas, une exécution séquentielle dans l'ordre des indices est imposée, mais le compilateur peut exploiter la structure de pipeline native dans la construction des dépendances interrépétition (*cf* Figure 29).

Dans une application, les dépendances entre les éléments d'une répétition se propagent à travers la hiérarchie et l'enchaînement des tâches et peuvent avoir un impact très négatif sur le parallélisme contenu dans l'application.

Sur l'exemple multidimensionnel de la Figure 31, la dépendance n'élimine pas tout le parallélisme. *Sur un espace de répétition multidimensionnel, une seule dépendance interrépétition ne peut pas éliminer complètement le parallélisme de la répétition.* En parcourant cet espace avec des hyperplans perpendiculaires sur le vecteur de dépendance, toutes les répétitions sur un de ces hyperplans sont des répétitions parallèles entre elles.

Même si le problème se complique quand sur un espace de répétition on a plusieurs dépendances, la solution peut être toujours représentée comme des hyperplans et des algorithmes pour les calculer sont disponibles. En utilisant ce type d'algorithmes, le compilateur peut exploiter les hyperplans contenant des répétitions parallèles et les aspects pipeline pour générer des ordonnancements optimisés.

L'aspect uniforme des dépendances interrépétition vise à limiter l'impact sur le parallélisme.

Sur l'exemple de la Figure 31, la dépendance avec un pas de 2 sur la dimension horizontale fait que sur l'espace de répétition 2D, des blocs successifs de deux colonnes contiennent des répétitions indépendantes, et donc parallèles. Le compilateur pourrait ordonnancer, en fonction des contraintes, les colonnes en blocs de deux colonnes parallèles, colonne par colonne ou même en séquentiel les éléments de chaque colonne, une colonne après l'autre.

2.6.4 *Dépendances à travers la hiérarchie*

Motivation

Dans la section précédente, nous avons vu comment exprimer des dépendances uniformes sur des espaces de répétition multidimensionnels. Comme discuté dans la section 2.4 et montré sur la Figure 2.4, pour déterminer l'espace de répétition d'une tâche il faut concaténer toutes les répétitions en descendant du niveau le plus haut jusqu'au niveau de la tâche. Une dépendance sur l'espace de répétition d'un niveau donne des dépendances entre des blocs de l'espace de répétition total. Afin de pouvoir exprimer des dépendances sur l'espace total de répétition, ou sur des parties de cet espace étalées sur plusieurs niveaux de la hiérarchie, la définition des dépendances interrépétition permet de les connecter à travers la hiérarchie.

Une raison encore plus forte est amenée par les transformations de boucles ARRAY-OL qu'on va discuter dans la section 5.3. Ces transformations sont des outils de refactoring de l'application, pour l'adapter au passage à un modèle d'exécution. Pour résumer, ces transformations ont un fonctionnement qui ressemble aux transformations de boucles usuelles appliquées sur des nids de boucles, avec la différence que dans le cas d'ARRAY-OL les transformations s'appliquent au niveau de la spécification sur des structures hiérarchiques de répétitions[20]. L'effet des transformations ARRAY-OL est la redistribution des répétitions à travers la hiérarchie, avec la création ou la suppression des niveaux de hiérarchie si besoin, et les transformations ARRAY-OL ont comme prémisse fondamentale l'invariance de la fonctionnalité.

[20] Contrairement aux nids de boucles au niveau de la compilation.

Définition 2 (fonctionnalités équivalentes)**.** Deux spécifications ont des fonctionnalités équivalentes si, pour toutes les valeurs d'entrées possibles, les valeurs de sortie sont les mêmes. ARRAY-OL est un langage d'expression des dépendances et donc pour que deux spécifications soient équivalentes il faut qu'elles expriment les mêmes dépendances.

Imaginons une transformation qui a comme effet la partition d'un espace de répétition en blocs, avec la génération d'un niveau de hiérarchie[21]. Pour que les spécifications restent équivalentes, une éventuelle dépendance interrépétition sur l'espace de répétition initial doit se traduire dans une structure qui exprime exactement les mêmes dépendances, mais maintenant sur l'ensemble des deux espaces de répétitions liées par le niveau de la hiérarchie.

[21] La transformation de *tiling* a cette fonctionnalité.

Connecter des dépendances des deux niveaux successives de la hiérarchie

La Figure 33 illustre la modélisation de l'exemple précédent avec une dépendance sur la première dimension après une transformation de *tiling* pour partager l'espace de répétition en blocs de 5×4 éléments[22]. Sur la modélisation cela se traduit par l'introduction d'un niveau de hiérarchie et le partage de l'espace de répétition entre ces deux niveaux ; l'espace de répétition total doit rester le même.

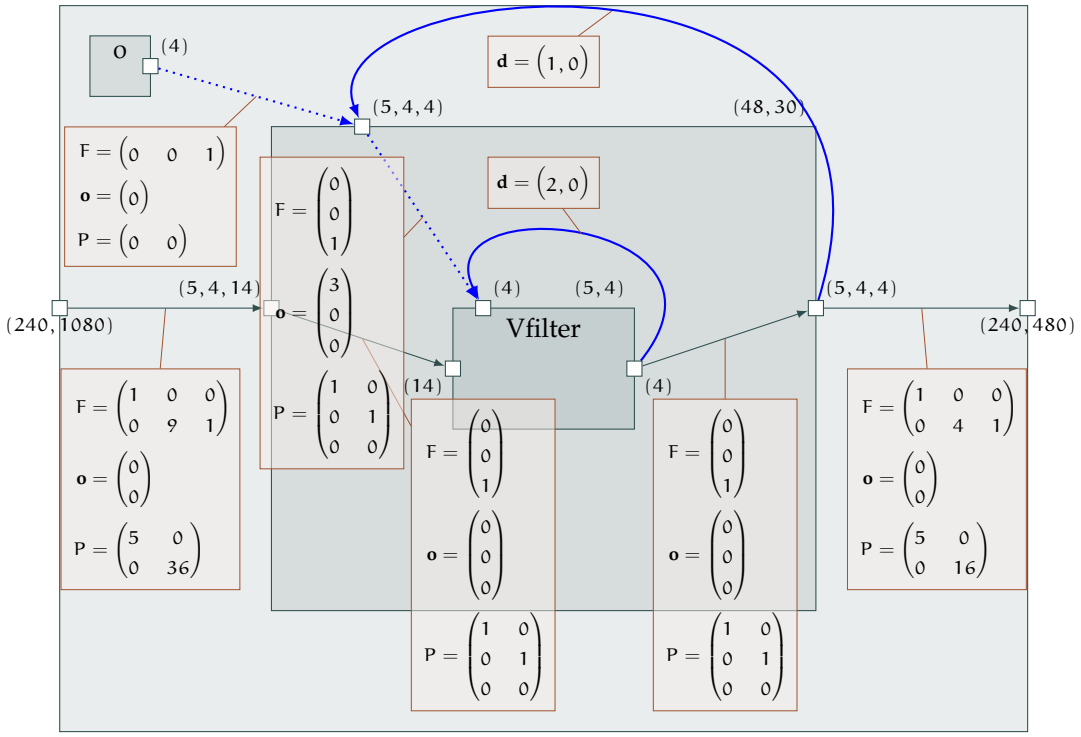
[22] Les deux modélisations ont des fonctionnalités équivalentes.

Après la transformation, la dépendance interrépétition initiale doit s'adapter à la nouvelle structure : la Figure 34 montre les dépendances dans l'espace de répétition partagé en blocs ; les nouvelles dépendances se trouvent davantage à l'intérieur des blocs et entre les éléments appartenant aux blocs voisins, mais en restant uniformes.

Ces dépendances se traduisent par une dépendance à l'intérieur des blocs et donc sur le niveau inférieur de la hiérarchie et une autre au niveau des blocs. En addition, pour exprimer la dépendance exacte entre des éléments dépendants des blocs différents, un lien par défaut ayant associé un tiler fait le lien entre les deux espaces de répétition. Le tiler est similaire au tiler de sortie du niveau inférieur de la hiérarchie avec une origine déplacée comme une différence entre la référence du bloc de dépendance et la référence de la fenêtre de dépendance – cette fenêtre est un bloc avec la même forme qui contient toutes les répétitions dépendantes pour ce bloc, comme montré sur la Figure 34.

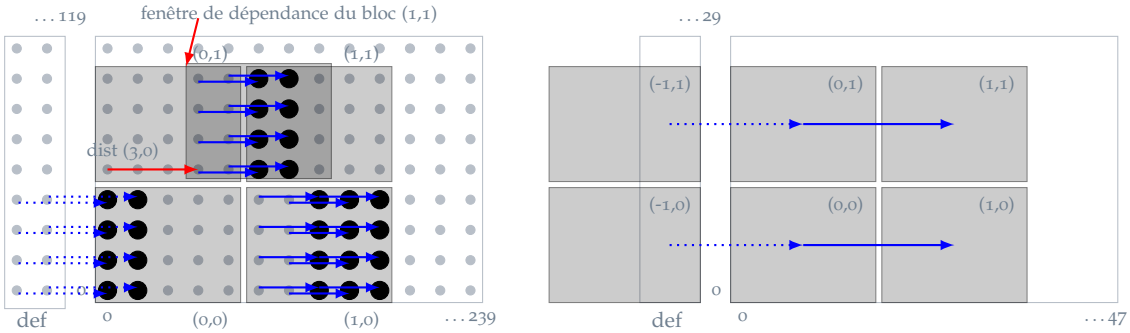
Détails sur la construction du tiler

Les similarités entre les tilers (le tiler de sortie et le tiler du lien par défaut) sont évidentes quand on analyse plus en détail la structure : les deux doivent avoir les mêmes formes pour les tableaux associés



L'espace de répétition a été partagé en $(48, 30)$ blocs de $(5, 4)$.

FIG. 33: La dépendance interrépétition après une transformation de *tiling*



Des parties de l'espace de répétition sont montrées sur la gauche de la figure : pour le bloc $(1, 0)$ toutes les répétitions qui dépendent des répétitions à l'intérieur du même bloc, pour le bloc $(1, 1)$ les répétitions qui dépendent des répétitions d'un bloc voisin et pour le bloc $(0, 0)$ les répétitions qui vont utiliser des valeurs par défaut. Sur la partie droite, les dépendances entre les blocs sont montrées.

FIG. 34: Dépendances après le partage en blocs

(les deux ports sont connectés par la dépendance du niveau haut), les mêmes formes pour les motifs associés (les deux ports sont connectés par la dépendance du niveau bas), et aussi le même espace de répétition.

Les blocs avec des valeurs par défaut

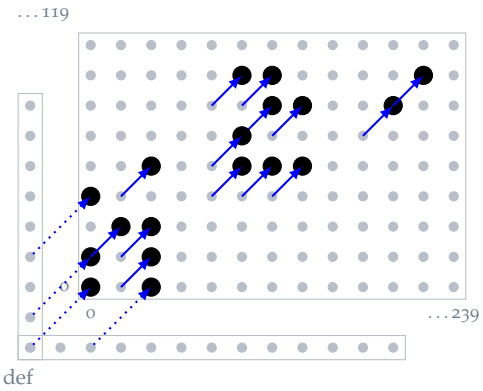
Pour les dépendances entre les blocs, le bloc $(0,1)$ et $(0,0)$ dépendent des blocs en dehors de l'espace de répétition et ainsi vont utiliser comme entrées des blocs remplis avec des valeurs par défaut représentées par des copies des valeurs par défaut fournies par le lien par défaut initial[23]. Le choix d'utiliser des blocs remplis avec des valeurs par défaut vient de la définition des dépendances interrépétition (Définition 1) qui contraint les deux ports de l'extrémité d'une telle dépendance à avoir la même forme. Les copies de la valeur par défaut initiale sont exprimées par le tiler associé qui construit un motif plus grand que le tableau par réplication. Les zéros des matrices du pavage et d'ajustage du tiler, en exceptant la dernière colonne d'ajustage, expriment que le tableau avec une dimension de (4) correspond à la dernière dimension du motif, alors que sur les autres dimensions d'ajustage et de pavage, le tableau est répliqué.

[23] Des copies virtuelles dans l'étape de la spécification ; afin de réduire les besoins en taille mémoire, au moment de l'exécution il n'y a pas besoin des copies mémoire.

2.6.5 *Dépendances multiples*

Une dépendance interrépétition spécifie la dépendance de données entre des couples de répétitions qui se trouvent à une distance égale au vecteur de dépendance dans l'espace de répétition. Et si on voulait exprimer qu'une répétition dépend de deux ou plusieurs autres ? En utilisant plusieurs dépendances interrépétitions sur le même espace de répétition, on peut exprimer des dépendances multiples, toujours uniformément espacées.

Prenant toujours le sous-ensemble du Downscaler (Figure 30) et en remplacent le vecteur de dépendance avec un vecteur de $(1,1)$ pour exprimer des dépendances en diagonale on se retrouve avec des dépendances comme montrées sur la Figure 35.

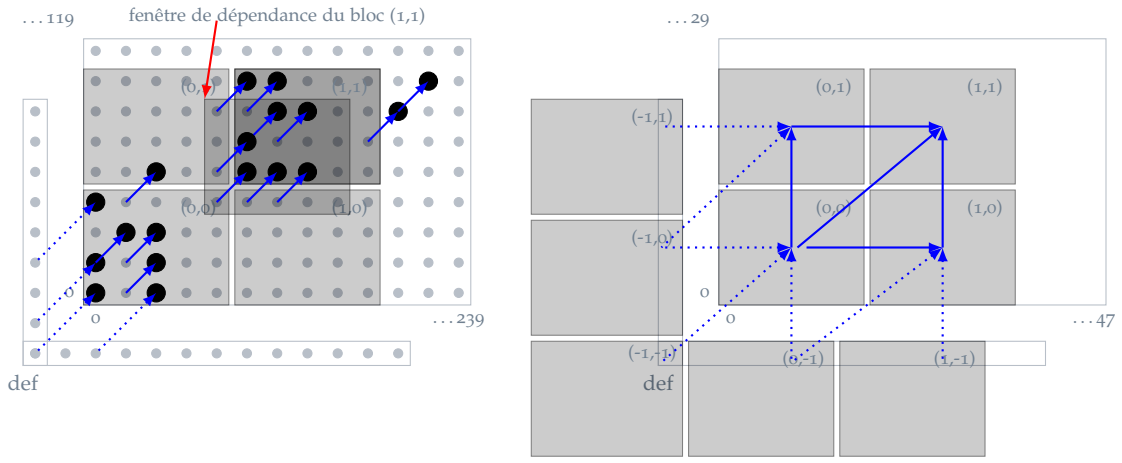


Des répétitions choisies aléatoirement sont montrées sur la figure.

FIG. 35: Dépendances entre les répétitions pour la dépendance diagonale

En partageant l'espace de répétition en blocs en utilisant la même transformation de tiling, les dépendances à l'intérieur d'un bloc et entre les blocs sont montrées sur la Figure 36.

La fenêtre de dépendance du bloc $(1,1)$ se croise avec trois autres blocs voisins et donc il y a trois dépendances entre les blocs associés aux trois liens par défaut au niveau bas de la hiérarchie.



Les dépendances diagonales après le partage en blocs sur la gauche. Des dépendances triples entre les blocs sur la droite.

FIG. 36: Dépendances diagonales après le *tiling* en blocs

La Figure 37 montre la représentation graphique de l'application en traitement par bloc, avec trois dépendances interrépétition au niveau du bloc.

La sélection du bloc approprié se fait en fonction des tilers associés aux trois liens par défaut ; les tilers sont toujours similaires l'un à l'autre et au tiler de sortie, avec l'exception de l'origine qui est déplacée. Des origines différentes assurent l'exclusivité de la sélection du lien par défaut : l'origine représente la différence entre la référence du bloc de dépendance et la référence de la fenêtre de dépendance. Avoir (1) la différence en valeur absolue entre quelconques deux origines qui dépasse, sur au moins une dimension, la taille du tableau (la même forme de tableau pour tous les tilers) et (2) la même matrice de pavage pour tous les trois tilers, garantissent que pour une répétition, un seul tiler va donner une référence valide.

DÉMONSTRATION DE L'EXCLUSIVITÉ DES TILER.

En présumant que pour une répétition $\mathbf{r}$ qui dépend d'une répétition en dehors de l'espace de répétition il y a un tiler (θ_ν) des liens par défaut qui fournit un élément de référence à l'intérieur des dimensions du tableau,

$$\mathbf{réf}_\nu = \mathbf{o}_\nu + P_\nu \cdot \mathbf{r} \tag{2.9}$$

et

$$\mathbf{0} \leqslant \mathbf{réf}_\nu < \mathbf{s}. \tag{2.10}$$

Pour tous les autres tilers, θ_i, on a

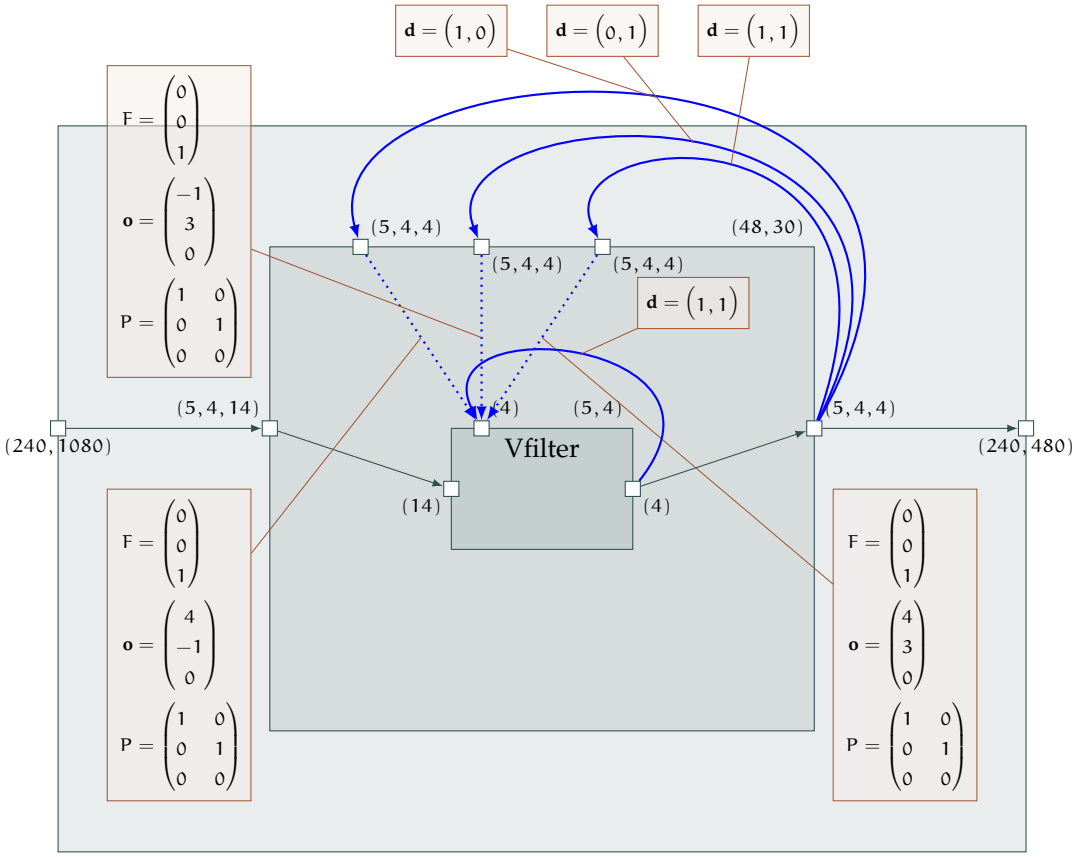
$$\forall\, i, 0 \leqslant i < t, i \neq \nu, \mathbf{réf}_i = \mathbf{o}_i + P_i \cdot \mathbf{r}, \tag{2.11}$$

des matrices identiques et les mêmes dimensions des tableaux, pendant que la différence entre les deux origines est δ_i :

$$P_i = P_\nu, \mathbf{o}_i = \mathbf{o}_\nu + \delta_i. \tag{2.12}$$

En remplaçant dans Équation 2.11, on se retrouve avec

$$\mathbf{réf}_i = \mathbf{o}_\nu + \delta_i + P_\nu \cdot \mathbf{r} = \delta_i + \mathbf{réf}_\nu. \tag{2.13}$$



La même application, la même transformation, la seule différence est que le vecteur de dépendance initial est un vecteur diagonal, $(1, 1)$. On a choisi de ne pas surcharger la figure avec des informations redondantes qui restent identiques avec la Figure 33 (tilers, lien par défaut) ; des modifications interviennent seulement au niveau des dépendances interrépétition.

FIG. 37: Dépendance diagonale

De l'hypothèse $|\delta_i| \not< \mathbf{s}$, soit $\mathbf{réf}_i \not\geqslant \mathbf{0}$ soit $\mathbf{réf}_i \not< \mathbf{s}$, qui signifie qu'aucun des tilers $\theta_i, i \neq v$, ne donne une référence valide et donc seulement le lien par défaut v est valide pour la répétition $\mathbf{r}$. □

2.6.6 *Discussion*

La spécification ARRAY-OL autorise des constructions plus complexes que SDF et ses extensions multidimensionnelles, avec l'exception des boucles de dépendance, exprimées en utilisant le concept de délai en SDF. Avec l'extension des dépendances interrépétition, on peut affirmer qu'ARRAY-OL est devenu un langage qui peut exprimer non seulement des délais et des états, mais des dépendances plus complexes dans les espaces de répétition et à travers la hiérarchie. Tout cela fait d'ARRAY-OL, dans notre opinion, un langage « complet » pour la spécification multidimensionnelle des applications de traitement intensif du signal.

De plus, les dépendances interrépétition sont capables d'exprimer des dépendances complexes connectées à travers les niveaux de la hiérarchie, extrêmement importantes pour ARRAY-OL dans le contexte des transformations ARRAY-OL, des outils clefs quand on parle de refactoring, optimisations et ordonnancement avec ARRAY-OL [3]. Ces transformations seront discutées en détail dans la section 5.3 et l'inter-

action avec les dépendances interrépétition est analysée et formalisée dans la chapitre 6.

Par l'enchaînement des dépendances interrépétition avec la structure hiérarchique d'Array-OL, on peut même construire des dépendances non uniformes, comme une répétition bidimensionnelle avec une dépendance linéaire sur toutes les lignes (chaque répétition dépend de la prétendante sur la même ligne), en temps que la première répétition de chaque ligne dépend de la dernière de la ligne précédente, pour exprimer une chaîne de dépendance en forme de Z.

Les dépendances interrépétition sont modélisées avec Array-OL en utilisant les mêmes concepts déjà disponibles – l'espace de répétition, les tilers (origine, ajustage, pavage) – et, dans cette démarche, très peu de nouveaux concepts ont du être ajoutés.

Une autre propriété très intéressante de SDF et de ses extensions est la possibilité d'ordonnancement statique des applications décrites. Comme déjà discuté en section 2.4, une spécification Array-OL qui respecte les règles de la définition statique est aussi ordonnancable statiquement. L'ordonnancement des nids de boucles avec des dépendances uniformes est un problème connu depuis les travaux de Darte et Robert [22]. En utilisant ces résultats, on peut ordonnancer statiquement des applications Array-OL avec des dépendances interrépétition.

Il s'agit de l'utilisation des techniques de recherche d'un ordonnancement linéaire qui respecte les dépendances uniformes par l'utilisation par exemple de la méthode de l'hyperplan proposée par Lamport [57]. La présence d'une seule dépendance interrépétition sur un espace de répétition détermine un ordonnancement sur des hyperplans perpendiculaires sur le vecteur de dépendance. Plusieurs dépendances sur le même espace de répétition compliquent l'algorithme de recherche, mais le problème peut toujours être projeté sur un problème d'optimisation linéaire.

2.7 CONCLUSIONS

Nous avons présenté dans ce chapitre le langage de spécification et modèle de calcul Array-OL, conçu pour la spécification des applications de traitement intensif de signal. Le langage vise l'expression complète de la régularité du calcul de ce type d'applications, qui se traduit par des décompositions en répétitions data-parallèles où maximum de parallélisme est exprimé et peut être exploité dans le contexte de l'exécution sur des plateformes embarquées multiprocesseurs.

Les contraintes dans le formalisme d'Array-OL pour le rendre ordonnancable statiquement (qui interdisaient de cycles dans le graphe de dépendances) nous ont conduit à proposer une extension pour l'expression des dépendances uniformes dans la spécification. Les dépendances interrépétition permettent l'expression de dépendances uniformes entre les répétitions à un niveau de hiérarchie et peuvent être connectées à travers la hiérarchie pour exprimer des dépendances entre les répétitions totales de la décomposition hiérarchique. En plus, le caractère uniforme de ces dépendances à un impact réduit sur le parallélisme dans la spécification.

Dans le chapitre suivant nous allons voir comment le formalisme d'Array-OL est utilisé dans le contexte de la comodélisation pour l'embarque en utilisant le standard MARTE d'OMG.

3

COMODÉLISATION DANS UNE VISION RÉPÉTITIVE EN MARTE

Dans ce chapitre nous allons nous intéresser à la comodélisation des systèmes en utilisant le standard OMG MARTE dans le contexte de l'environnement GASPARD2.

Tous les concepts définis pour le langage ARRAY-OL sont disponibles dans le standard et nous allons voir comment une spécification en MARTE peut être réduite à une spécification ARRAY-OL et donc toutes les propriétés dérivées du modèle de calcul ARRAY-OL s'appliquent à une spécification MARTE. Dans cette direction, on peut profiter des outils de conception autour UML, de l'ingénierie dirigée par les modèles, d'avoir une ouverture facile vers la communauté apportée par le standard OMG et en même temps tous les avantages du modèle de calcul, tel que la définition statique du langage.

En plus de la modélisation des applications, la décomposition répétitive permet la modélisation compacte des architectures avec des topologies régulières et facilite l'exploration du placement des applications répétitives sur telles architectures.

Nous commençons, dans la section 3.1, par la définition des concepts de l'IDM que nous allons utiliser par la suite pour présenter le standard MARTE (dans la section 3.2), en insistant sur les concepts qui permettent modéliser des structures répétées, section 3.3. La section 3.4 décrit une construction répétitive appelée *reshape* permettant l'expression de relations uniformes entre deux formes multidimensionnelles.

L'environnement GASPARD2 est présenté dans la section 3.5 et nous allons voir comment les concepts répétitifs de MARTE peuvent être

utilisés pour la modélisation des applications, des architectures et du placement répétitif, respectivement dans les section 3.6, section 3.7 et section 3.8.

3.1 CONCEPTS SUPPORT

Avant de continuer, nous devrons introduire un ensemble de concepts clés dans le contexte de l'ingénierie dirigé par des modèles (IDM) que nous allons utiliser dans ce chapitre :

LES MODÈLES sont la base de la récente méthodologie d'ingénierie dirigée par les modèles, mais le concept de modèle en lui-même est très ancien comme le dit Favre dans [34]. Ainsi, toute représentation peut être considérée comme un modèle. Un modèle est une vue abstraite d'un système informatique réalisée selon un (ou plusieurs) point(s) de vue. Pour réaliser un modèle, il est important d'identifier les caractéristiques pertinentes et intéressantes du système afin de les modéliser et ainsi de permettre l'étude du système selon ces caractéristiques. L'IDM repose sur une utilisation intensive des modèles tout au long du processus de développement.

LES MÉTAMODÈLES ont le rôle d'ajouter une sémantique et une syntaxe aux modèles afin qu'ils soient compréhensibles par les machines. Un modèle fait dans un métamodèle est dit *conforme à* ce métamodèle. Il est possible de faire une analogie entre les métamodèles et les grammaires des langages de programmation. Un métamodèle doit rendre possible la modélisation de tou tes les caractéristiques pertinentes du type de systèmes à modéliser.

UNIFIED MODELING LANGUAGE (UML) est un langage de modélisation graphique standardisé par l'OMG [72]. Ce langage est basé sur un métamodèle auquel est associée une représentation graphique. Le principal inconvénient de ce langage est qu'il ne propose une sémantique faible, une des raisons pour laquelle la notion de *profil* a été introduite afin d'ajouter une sémantique à un modèle UML par l'intermède des *stéréotypes*. Un stéréotype ajoute une sémantique à l'élément UML sur lequel il est placé, et peut également ajouter des attributs à cet élément à l'aide des *tagged values*(balises). Ce mécanisme d'extension d'UML est à la base de son succès. Une autre clé de son succès est que de nombreux outils proposent une implémentation de ce standard. Il s'agit aussi bien d'outils propriétaires (MagicDraw UML[1], Enterprise Architect[2]) que d'outils libres tels que Papyrus UML[3].

TRANSFORMATIONS DE MODÈLES. Rendre un modèle *productif* entraîne qu'il faut, comme son nom l'indique, produire. Les transformations de modèles [63] permettent cette production. Une transformation prend un ou plusieurs modèles d'entrée (conformes à leurs métamodèles respectifs) et produit un ou plusieurs modèles de sortie (également conformes à leurs métamodèles respectifs). Le passage des modèles sources vers les modèles cibles est décrit à l'aide de *règles de transformations* qui sont exécutées par un *moteur de transformation*. Il existe de nombreux outils permettant la description et l'exécution de transformation de modèles (Kermeta [88, 67], ATL [48], ModelMorf [30],etc.). Ces outils ne sont pas standard et sont donc très spécifiques. Le seul standard

[1] http://www.nomagic.com
[2] http://www.sparxsystems.com.au/
[3] http://www.papyrusuml.org/

de transformation de modèle est Query/View/Transformation (QVT) [75] standardisé par l'OMG.

3.2 MARTE : MODELING AND ANALYSIS OF REAL-TIME AND EMBEDDED SYSTEMS

La conception système basé sur une approche orientée modélisation haut-niveau devient de plus en plus attractive. Les concepteurs peuvent atteindre des niveaux plus hauts d'abstraction en se concentrant sur un domaine particulier, en profitant d'une forme d'expression visuelle qui facilite la compréhensibilité du système et améliore la communication entre concepteurs. Les concepts et les relations à travers lesquels le modèle d'un système est décrit sont spécifiés précisément dans le métamodèle auquel le modèle est conforme. Afin de disposer d'une interface graphique pour la représentation et la manipulation de ces modèles, deux alternatives sont possibles :

A. définir une vue graphique pour chaque concept du métamodèle ;
B. élargir UML avec un profil pour supporter ces concepts.

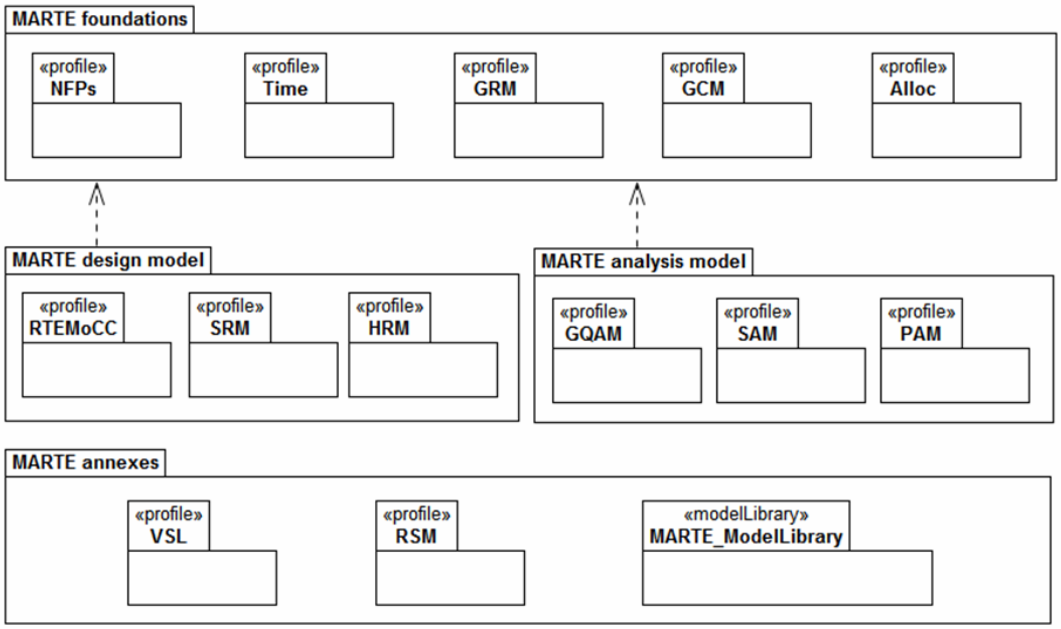
Même si des outils comme Eclipse, EMF, GMF facilitent énormément la tâche, la première solution reste relativement fastidieuse due à la nécessité d'avoir un outil dédié. La deuxième solution vise à ajouter des sémantiques et des nouveaux concepts à UML, dans le but de l'adapter à un domaine spécifique et d'utiliser des outils de conception tels que MagicDraw UML, Enterprise Architect ou Papyrus UML.

MARTE (Modeling and Analysis of Real-time and Embedded systems) [71, 83, 79, 80] est un profil standard d'UML proposé par l'OMG (Object Management Group), visant principalement à ajouter à UML des capacités pour le développement dirigé par des modèles des systèmes embarqués temps réel. UML fournit le cadre pour brancher les concepts nécessaires. Le profil MARTE étend les possibilités pour la modélisation applicative, architecturale et des relations entre elles. Il permet aussi des extensions pour faire l'analyse des performances et d'ordonnancement, de même que prendre en considération des services plateforme (telles que les services fournis par un système d'exploitation).

La Figure 38 présente l'architecture globale et la décomposition en paquetages du profil MARTE. Le profil est structuré suivant deux directions, la modélisation des concepts des systèmes embarqués temps-réel et l'annotation des modèles d'applications pour supporter l'analyse des propriétés système. L'organisation du profil reflète cette structure, par la séparation des deux paquetages, *MARTE design model* et *MARTE analysis model*, respectivement.

Ces deux parties majeures partagent des concepts communs, groupés dans *MARTE foundations* : pour exprimer des propriétés non fonctionnelles (NFPs), des notions de temps (Time), la modélisation des ressources (GRM), des composants (GCM) et des éléments d'allocation (Alloc). Un paquetage additionnel contient les annexes du profil définies en MARTE, ainsi bien que des librairies prédéfinies.

L'annexe *Repetitive Structure Modeling* (RSM) de MARTE est inspirée du modèle de calcul Array-OL. Tous les concepts utilisés en Array-OL, décrits dans chapitre 2 sont présents et leur sémantique est équivalente à celle d'Array-OL.



NFPs = Non-Functional Properties, **GRM** = Generic Resource Modeling, **GCM** = Generic Component Model, **Alloc** = Allocation modeling, **RTEMoCC** = RTE Model of Computation & Communication, **SRM** = Software Resource Modeling, **HRM** = Hardware Resource Modeling, **GQAM** = Generic Quantitative Analysis Modeling, **SAM** = Schedulability Analysis Modeling, **PAM** = Performance Analysis Modeling, **VSL** = Value Specification Language, **RSM** = Repetitive Structure Modeling.

FIG. 38: Architecture globale du profil MARTE

3.3 MODÉLISATION DES STRUCTURES RÉPÉTÉES

L'extension RSM de MARTE est destinée à proposer des modalités pour l'expression compacte des systèmes ayant une structure ou une topologie répétitive. Ces structures considérées sont décomposées en répétitions de sous-composants, interconnectés via des motifs réguliers.

Les mécanismes disponibles sont orientés vers deux aspects :

A. la possibilité de spécifier la forme (*shape*) d'une répétition par une forme multidimensionnelle, et ainsi permet la représentation d'une collection de liens potentiels comme un tableau multidimensionnel. Le gain est double : facilite l'expression des liens de topologie et augmente le pouvoir d'expression du mécanisme de description de la topologie ;

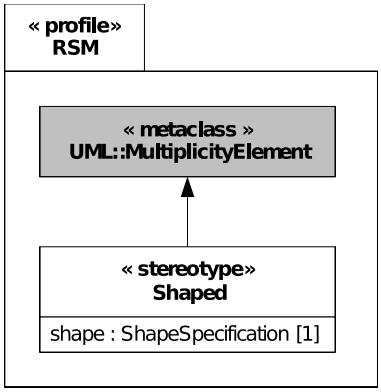
B. une modalité de spécification des informations topologiques sur des relations entre des entités, pour exprimer des liens topologiques entre des entités au moment de l'exécution.

Ces mécanismes peuvent être utilisés pour spécifier :

1. des architectures matérielles d'exécution : pour exprimer tout le parallélisme disponible, précisément et sous une forme compacte ;
2. modélisation d'applications : pour exprimer le maximum de parallélisme disponible de l'application (parallélisme de tâches et de données) ;
3. un placement régulier, temporel et spatial, de l'application sur des plateformes d'exécution matérielles.

Concernant les concepts additionnels, il s'agit de :

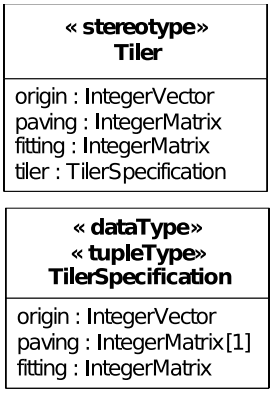
A. l'extension du concept de multiplicité d'UML pour le cas multidimensionnel, représenté par le stéréotype *Shaped* (Figure 39) ;



Le diagramme UML définit le stéréotype « Shaped » comme l'extension de la métaclasse *UML::MultiplicityElement* avec la balise *shape* définissant une forme multidimensionnelle représentée par le datatype *ShapeSpecification*.

FIG. 39: *Shapes* multidimensionnelles

B. Les types de données *ShapeSpecification* et *TilerSpecfication* (Figure 40 :



Le diagramme UML définit les types de données :
- *TilerSpecfication* comme une collection d'un vecteur d'origine et deux matrices, d'ajustage et de pavage, qui définissent le concept de tiler ;
- *ShapeSpecification* comme une liste ordonnée de valeurs naturelles définissant la taille de chaque dimension de la forme multidimensionnelle.

FIG. 40: TilerSpecification et ShapeSpecification

C. l'extension des liens de connexion UML (*Connector*) pour exprimer des liens de topologie répétitifs. La Figure 41 montre ces liens ; les concepts de *Tiler*, *InterRepetition* et *DefaultLink* sont équivalents à la spécification ARRAY-OL et le lien de *Reshape* permet l'expression de liens de dépendance par motif entre deux tableaux multidimensionnels.

Nous allons présenter par la suite le concept de *Reshape* permettant l'expression de relations entre des motifs uniformes entre deux tableaux multidimensionnels.

3.4 RESHAPES

L'expression de dépendances régulières entre les éléments de deux tableaux multidimensionnels est une construction essentielle en MARTE. Le concept de *Reshape* permet la spécification d'un réarrangement des motifs tuilés différemment dans les tableaux source et destination : création/suppression des dimensions, linéarisation/scission des dimensions, duplication des éléments.

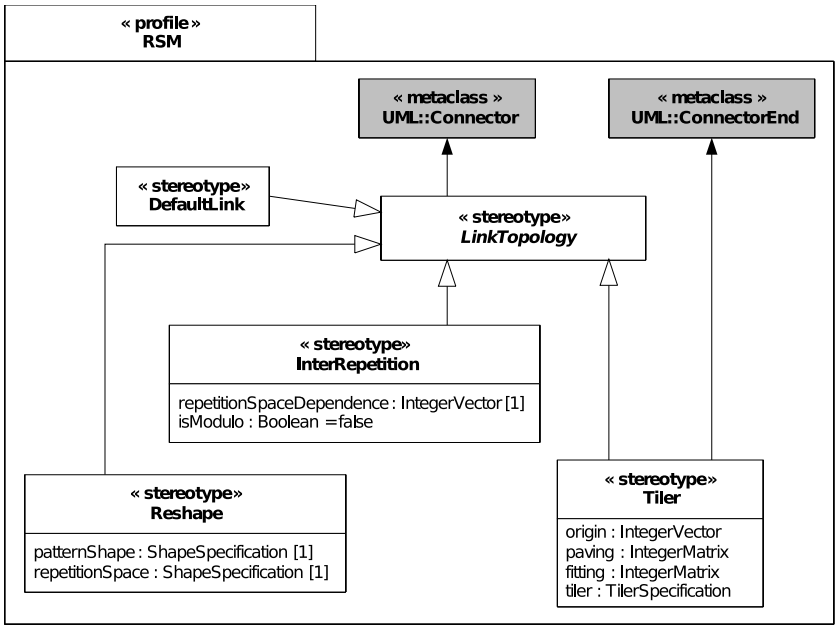


FIG. 41: Liens topologiques répétitives

Cette construction peut être vue comme une représentation compacte d'un cas spécial de répétition et elle est essentielle pour la comodélisation en MARTE, permettant l'expression des :

- restructurations des tableaux multidimensionnels dans l'application ;
- liens « moins » réguliers dans la modélisation de l'architecture ;
- distributions uniformes de l'application sur l'architecture.

Le concept de *Reshape* exprime le réarrangement des éléments entre deux tableaux multidimensionnels. Dans le contexte de la spécification applicative, ces constructions peuvent être utilisées pour exprimer un lien d'égalité entre les éléments d'un tableau source, caractérisé par sa forme $\mathbf{s_s}$, et un tableau cible, avec sa forme $\mathbf{s_c}$, par la scission les éléments du tableau source dans une série de $\mathbf{s_R}$ motifs identiques avec la forme $\mathbf{s_M}$, en utilisant en tiler source θ_s, et par le réarrangement des ces motifs dans le tableau destination, eu utilisant un tiler cible θ_c.

Le concept peut être vu comme une représentation compacte d'une répétition d'une tâche *Égalité*, comme le montre Figure 42. Une représentation compacte est plus appropriée pour l'expression du réarrangement des données, en montrant qu'il n'y a pas de calculs.

3.4.1 *Spécification formelle*

Définition 3 (connecteur de reshape)**.** Un connecteur reshape peut être utilisé pour remplacer un connecteur d'assemblage[4] d'une tâche composée, avec la différence que les deux ports connectés peuvent avoir des formes différentes $\mathbf{s_s}$ et $\mathbf{s_c}$[5] et la présence des informations de reshape, $\rho = (\mathbf{s_R}, \mathbf{s_M}, \theta_s, \theta_c)$:

- la répétition des motifs, caractérisé par une forme $\mathbf{s_R}$;
- la forme des motifs, $\mathbf{s_M}$;
- le tiler source $\theta_s = (F_s, \mathbf{o_s}, P_s) \in \mathbb{Z}^{\dim(\mathbf{s_s})\times\dim(\mathbf{s_M})} \times \mathbb{Z}^{\dim(\mathbf{s_s})} \times \mathbb{Z}^{\dim(\mathbf{s_s})\times\dim(\mathbf{s_R})}$;

[4] entre deux ports appartenant à deux appels de tâche.

[5] Différemment au connecteur normal où les deux ports doivent obligatoirement avoir la même forme.

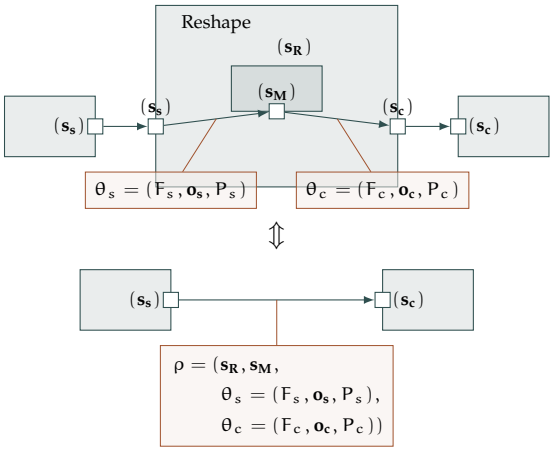


FIG. 42: Reshape comme une répétition

– le tiler cible $\theta_c = (F_c, \mathbf{o_c}, P_c) \in \mathbb{Z}^{\dim(\mathbf{s_c})\times\dim(\mathbf{s_M})} \times \mathbb{Z}^{\dim(\mathbf{s_c})} \times \mathbb{Z}^{\dim(\mathbf{s_c})\times\dim(\mathbf{s_R})}$.

Définition 4 (relation d'égalité entre les éléments des ports connectés par un reshape)**.** Un connecteur de reshape $((s, d_s, \tau_s, \mathbf{s_s}), (c, d_c, \tau_c, \mathbf{s_c}), \rho)$ défini par la Définition 3, spécifie les relations suivantes entre les éléments de s et c

$$\begin{cases} \forall \mathbf{r} \in \mathbb{N}^{\dim(\mathbf{s_R})}, \mathbf{0} \leqslant \mathbf{r} < \mathbf{s_R} \\ \forall \mathbf{i} \in \mathbb{N}^{\dim(\mathbf{s_M})}, \mathbf{0} \leqslant \mathbf{i} < \mathbf{s_M} \\ \mathbf{j_s} = \mathbf{o_s} + (P_s \; F_s) \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s_s} \\ \mathbf{j_c} = \mathbf{o_c} + (P_c \; F_c) \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s_c} \\ s[\mathbf{j_s}] \leftrightarrow c[\mathbf{j_c}] \end{cases}, \tag{3.1}$$

où $\leftrightarrow$ représente la relation de connexion entre les éléments et peut être interprétée selon le contexte :

– dans l'application, la relation exprime une dépendance de données ;
– dans l'architecture, un lien physique ;
– dans l'allocation, une association.

3.4.2 *Impact sur l'ordre partiel de la tâche composée*

Un connecteur simple d'une tâche composée spécifie une dépendance un à un entre les éléments des deux tableaux (ports) connectés, qui détermine une dépendance de données entre les appels qui contient les deux ports. Une construction de reshape spécifie des dépendances par motifs entre les éléments des deux ports connectés, et donc la dépendance entre les appels persiste. La présence d'un reshape sur le lien du graphe d'une tâche composée ne change rien en matière de dépendances de données entre les appels et, en conséquence, d'ordre partiel strict au niveau de la tâche composée.

Observation. Dans la description d'une application, des contraintes sur la spécification du reshape sont héritées de sa forme répétitive. Pareil

que pour une tâche répétée, le tiler cible doit produire entièrement le tableau destination et les tuiles doivent ne pas se chevaucher.

Par la suite nous allons présenter l'environnement de conception GASPARD2 et comment le profil MARTE est utilisé dans l'étape de spécification.

3.5 GASPARD2 : ENVIRONNEMENT DE COMODÉLISATION POUR L'EMBARQUÉ

GASPARD2 (Graphical Array Specification for Parallel and Distributed Computing) [20, 79] est un environnement de développement intégré (IDE[6]) dédié à la comodélisation visuelle des systèmes embarqués, développé au sein de l'équipe DaRT de l'INRIA LILLE-NORD EUROPE et du Laboratoire d'Informatique Fondamentale de Lille (LIFL). En utilisant une méthodologie IDM, il vise la modélisation, la simulation, l'analyse et la génération du code pour des applications et des architectures matérielles SoC. Il permet la représentation conjointe des parties applicatives et architecturales d'un système. Un mécanisme d'association est disponible pour le placement des applications sur des architectures.

[6] De l'anglais Integrated Development Environment.

À partir cette modélisation, une série de transformations de modèles permet, en passant par différents niveaux d'abstraction, la spécialisation de ce système pour différentes cibles :

- *validation* avec des langages synchrones [93] ;
- *exécution* en OpenMP (C ou Fortran) [86] ;
- *simulation* au niveau TLM (Transaction Level Modeling) [17] en SystemC [9, 78] ;
- *synthèse* FPGA en passant par du code VHDL [58].

L'architecture de GASPARD2 est schématisé sur le diagramme de la Figure 43.

En commençant avec la spécification conjointe en MARTE de l'application, de l'architecture, du placement et le déploiement des composants élémentaires sur des fonctions de bibliothèque, le modèle du système est traduit successivement entre plusieurs niveaux d'abstraction, chacun correspondant à un métamodèle qui ajoute des propriétés spécifiques, jusqu'à la génération du code.

Exemple. Pour la branche *SystemC*, le modèle UML est traduit dans une représentation équivalente conforme au métamodèle MARTE, en allégeant le modèle des concepts non utilisés et de la représentation graphique. Ensuite, les répétitions distribuées sont traduites dans des polyèdres et un outil externe est appelé pour la génération des boucles qui vont parcourir correctement les polyèdres (il s'agit de l'outil *CLooG* [1] – Chunky Loop Generator). Les modules SystemC qui respectent la spécification (dépendances de données, placement, etc.) sont générés à partir ce dernier métamodèle et peuvent être utilisés pour la simulation du système.

GASPARD2 a fortement contribué au développement du profil UML standard OMG de MARTE. Le paquetage RSM – Repetitive Structure Modeling – de MARTE est inspiré de GASPARD2 et son modèle de calcul, ARRAY-OL. Le paquetage HRM – Hardware Resource Modeling – utilise aussi des concepts architecturaux pris de GASPARD2.

Dans le but de rendre l'environnement GASPARD2 plus « standard », pour faciliter l'interfaçage avec d'autres outils et de profiter des outils

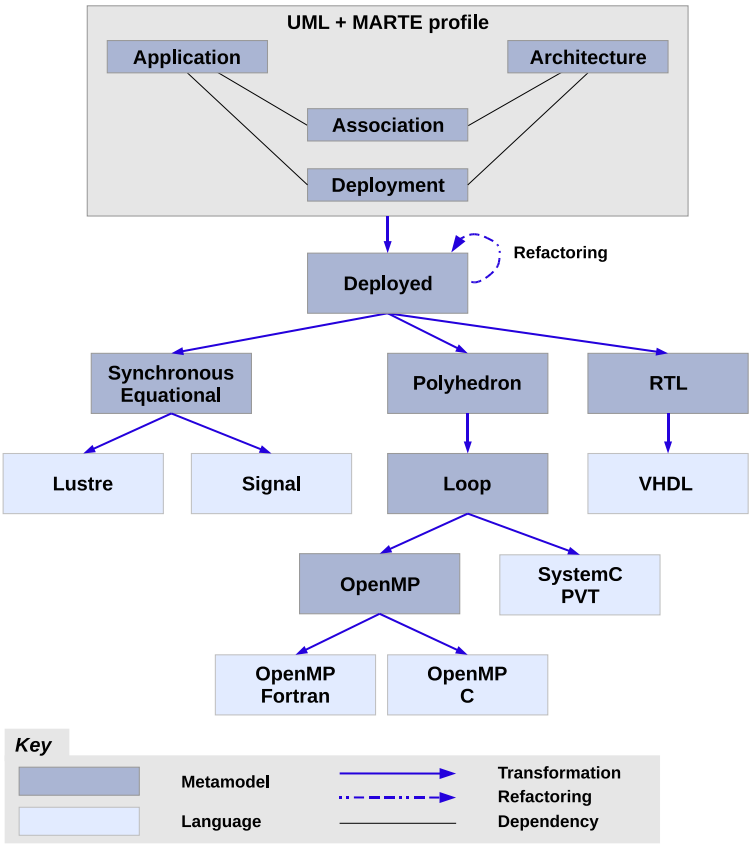


FIG. 43: La méthodologie SoC de GASPARD2

et technologies disponibles autour, des efforts ont été faits pour rendre GASPARD2 complètement compatible avec le MARTE ; au moment de l'écriture de ce document seuls les concepts de déploiement ne sont pas intégrés en MARTE.

3.6 MODÉLISATION EN MARTE POUR GASPARD2

Au début de ces travaux de thèse, le concepteur disposait d'un profil GASPARD2 [10] pour la spécification et la manipulation des modèles, en profitant des outils de modélisation UML, notamment MagicDraw UML. Le profil GASPARD2 a pour origine les travaux de Cuccuru lors de sa thèse [19] et est basé sur le langage ARRAY-OL présenté précédemment.

Une transformation de modèle permettait l'import des modèles (application, architecture, association et déploiement) dans l'environnement GASPARD2 basé sur Eclipse[7], sous la forme des modèles GASPARD2 conformes à un métamodèle GASPARD2. Les concepts du profil et de métamodèle GASPARD2 sont équivalentes avec la différence que les concepts UML non utilisés ont été enlevés pour simplifier la modélisation.

[7] http://www.eclipse.org/

Avec l'apparition du standard MARTE, des efforts ont été faits dans l'équipe pour rendre l'environnement complètement compatible avec le standard et supprimer le passage par le métamodèle GASPARD2, une étape redondante vue que MARTE a été influencé par GASPARD2 et que presque tous les concepts se retrouvent dans le standard.

En dehors des concepts de déploiement, toutes les autres concepts de l'ancien profil GASPARD2 *se retrouvent dans le profil* MARTE.

[8] http://www.papyrusuml.com/

Le nouvel environnement GASPARD2 profite aussi du développement de l'éditeur graphique Papyrus UML[8], intégré à Eclipse et open source. Dans cette démarche, tout le développement GASPARD2 actuel, de la spécification à la génération du code, est structuré autour la plate-forme Eclipse et des extensions peuvent être facilement développées et branchées au noyau GASPARD2. Figure 44 montre une capture d'écran de l'environnement, la modélisation en Papyrus au niveau de la spécification. Pour décrire la de-composition hiérarchique tâches/sous-tâches, en Papyrus nous utilisons le diagramme *composite* d'UML.

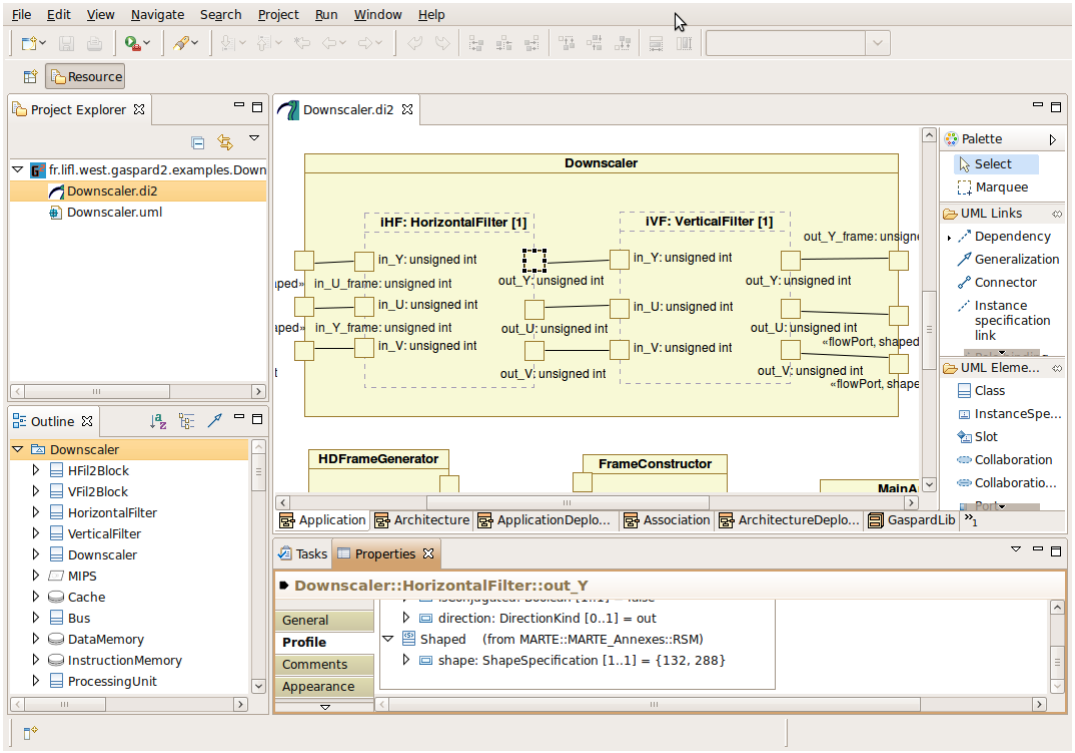


FIG. 44: L'environnement GASPARD2

3.6.1 *Équivalence des concepts*

Le modèle de calcul utilisé par l'environnement est ARRAY-OL et nous avons vu dans le chapitre 2 les bases de la sémantique du langage, et aussi les extensions pour exprimer des dépendances uniformes dans le section 2.6.

Pour faciliter la conception en utilisant des outils de modélisation visuels, en GASPARD2 on utilise des modèles UML étendus par le profil MARTE, où on retrouve tous les concepts d'ARRAY-OL.

Par la suite nous allons voir comment on modélise avec MARTE des systèmes qui correspondent au modèle de calcul d'ARRAY-OL et quels sont les concepts équivalents, mais avant il faut pointer quelques observations très importantes :

VALIDITÉ DE LA MODÉLISATION. Une modélisation en MARTE pour GASPARD2 est considérée valide si elle correspond à une définition ARRAY-OL, et qu'en plus cette définition respecte les règles de la définition statique ARRAY-OL. De plus, pour réduire la complexité de la modélisation, une spécification en MARTE est aussi

valide *si elle peut être réduite à une définition* ARRAY-OL *valide*, par des transformations structurales que nous allons discuter dans la sous-section 3.6.2 ;

MODÉLISATIONS NON-APPLICATIVES. Les concepts du modèle de calcul ARRAY-OL sont définis dans la vision d'une modélisation applicative. GASPARD2 permet la modélisation conjointe de l'application, l'architecture et le placement de l'application sur des structures architecturales. Dans ce sens, certains concepts ARRAY-OL, même si la syntaxe reste la même, changent légèrement de signification, discuté par la suite dans la sous-section 3.7.3.

Les concepts ARRAY-OL *ont des concepts équivalents (homonymes) en* UML *plus le profil* MARTE.

Le métamodèle UML convient parfaitement à une spécification équivalente à ARRAY-OL par les concepts autour sa vue *Composite* : une décomposition hiérarchique en sous-composantes interconnectées par leurs ports :

1. les composantes ARRAY-OL correspondent aux classes UML – *Class* ;
2. les ports ARRAY-OL correspondent aux ports d'UML – *Port* – étendus par le stéréotype *FlowPort* pour spécifier la direction et le stéréotype *ShapeSpecification* pour la forme multidimensionnelle du tableau ;
3. les types des ports, représentant le type des données des tableaux associés aux ports, peuvent être des types primitifs d'UML, MARTE ou des types composés ou primitifs définis en UML ;
4. les sous-tâches correspondent au propriétés – *Property* – avec le type de la sous-composante et le stéréotype *ShapeSpecification* pour exprimer l'espace de répétition, si besoin[9] ;
5. les connecteurs ARRAY-OL correspondent aux connecteurs UML – *Connector* – étendus par les stéréotypes du paquetage RSM pour désigner les tilers (stéréotype *Tiler*), les dépendances interrépétitions (stéréotype *InterRepetition*), les liens par défaut (stéréotype *DefaultLink*) ou les reshapes (stéréotype *Reshape*).

[9] Pour des sous-tâches répétées.

Observation. Les stéréotypes MARTE ajoutent des concepts aux éléments UML, des balises (*tagged values* en anglais), permettant la spécification complète des modèles ; par exemple, le stéréotype *Tiler* ajoute les concepts d'origine, ajustage et pavage aux connecteurs UML.

La modélisation multidimensionnelle nécessite la spécification des vecteurs et des matrices pour les liens de la structure répétitive et des *shapes* pour la multidimensionnalité des tableaux et des répétitions :

SHAPESPECIFICATION. La multiplicité d'UML, définie comme un intervalle inclusif d'entiers non négatifs commençant avec une borne inférieure et finissant avec une borne supérieure potentiellement infinie, est associé au concept abstrait de *MultiplicityElement*, commun à de nombreux d'éléments UML, incluant *Property* et *Port*. Le concept est étendu par MARTE à une multiplicité multidimensionnelle via une *ShapeSpecification*. Une shape est définie par une liste ordonnée d'entiers non nuls, où au plus une dimension est infinie. Les éléments de la liste sont entourés par des accolades et séparés par des virgules, pendant que l'infini est représenté par le caractère étoile ;

Exemple. Un tableau de données tridimensionnel représentant un flot infini d'images de taille 1920×1080 est défini par la shape de $\{1920, 1080, *\}$.

INTEGERVECTOR. Le type de vecteur de valeurs entiers est défini dans la bibliothèque MARTE (*cf* la Figure 45) comme une liste ordonnée des valeurs entiers (possiblement zéro) et est défini par la liste d'entiers entourés par des accolades et séparés par des virgules ;

INTEGERMATRIX. Une matrice des valeurs entiers (*cf* la Figure 45) est représentée par une liste des vecteurs entiers, qui respectent la définition précédente et qui ont la même taille.

Observation. Dans la spécification MARTE, les matrices sont écrites colonne par colonne. Par exemple, une matrice de $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ est représentée par $\{\{1,3,5\},\{2,4,6\}\}$.

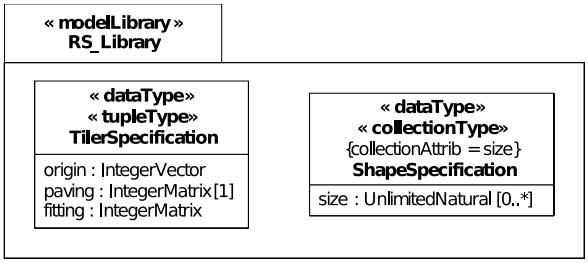


FIG. 45: Les types de données vecteur et matrice

Observation. La spécification des shapes ne peut pas contenir des valeurs nulles, pendant que les vecteurs et les matrices ne peuvent pas contenir des valeurs infinies.

Exemple. Dans l'Appendice A on peut retrouver la modélisation en MARTE de l'application de fenêtres glissantes modélisée avant en ARRAY-OL. Sur les deux figures on peut clairement identifier les similitudes entre une modélisation en MARTE et la spécification ARRAY-OL.

Nous allons nous intéresser par la suite à la validité d'une spécification en MARTE pour GASPARD2.

3.6.2 *Règles de réduction vers une définition valide*

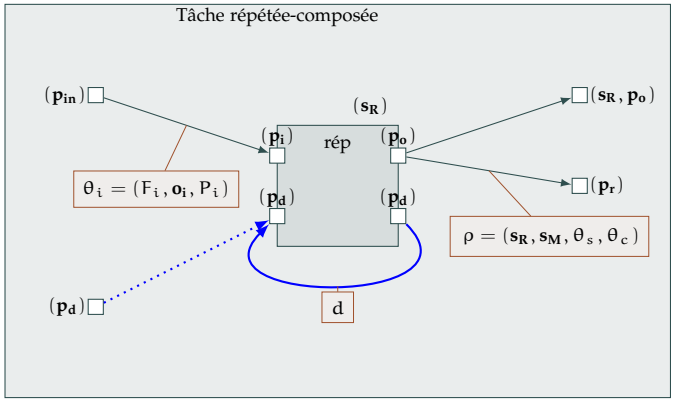
Une modélisation GASPARD2 en utilisant UML et le profil MARTE est valide dans la sémantique de GASPARD2 si elle correspond à une description ARRAY-OL correcte. Une description ARRAY-OL peut être dérivée à partir de la modélisation en MARTE par des transformations un à un entre les concepts équivalents.

Néanmoins, ce choix fait que toutes restrictions du modèle de calcul ARRAY-OL doivent être respectées dans la modélisation en MARTE et détermine une complexité non nécessaire de point de vue de l'utilisateur. Le cas le plus souvent rencontré en pratique est causé par l'exclusion entre les structures composées et répétitives d'une tâche, qui fait que chacune des répétitions ARRAY-OL doit contenir une seule tâche répétée. Pour faciliter la tâche de l'utilisateur et réduire la complexité de la description hiérarchique, nous avons choisi de permettre des modélisations simplifiées qui sont correctes même si elles ne respectent exactement la définition ARRAY-OL, mais des règles précises de transformation peuvent transformer une telle modélisation vers une description ARRAY-OL correcte.

TÂCHE RÉPÉTITIVE-COMPOSÉE

Une tâche ARRAY-OL peut être soit élémentaire, soit répétitive (une seule sous-tâche répétée), soit composée (potentiellement plusieurs sous-tâches, toutes non répétées). Dans la modélisation MARTE, la coexistence des sous-tâches répétées et non répétées dans une décomposition est permise, si la spécification permet la transformation vers une description ARRAY-OL suite à une règle de transformation qui dit que, si la décomposition est de type répétitive-composé[10], toutes les sous-tâches répétées sont transformées par l'introduction d'une tâche répétée intermédiaire par sous-tâche répétée dans la hiérarchie et la transformation de tous les liens connectés à cette sous-tâche.

[10] Une tâche est répétitive-composée si elle contient au moins deux sous-tâches et au moins une des sous-tâches est répétée.



Les ports qui n'appartient à aucune tâche expriment qu'ils peuvent appartenir à une tâche quelconque dans la description composée (des sous-tâches ou même de la tâche répétée-composée).

FIG. 46: Tâche répétitive-composée

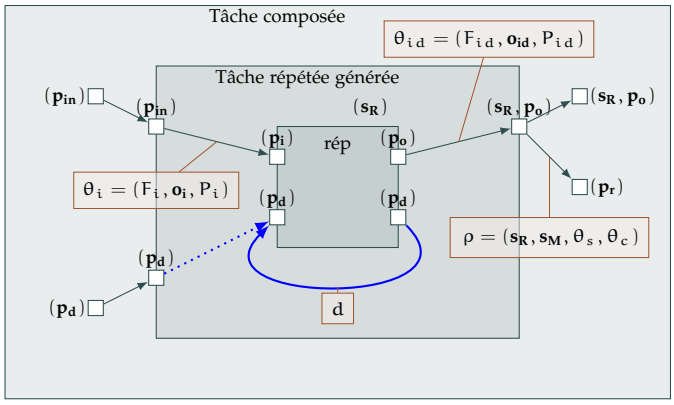


FIG. 47: Tâche répétitive-composée transformée

Un exemple est illustré sur la Figure 47, montrant la transformation de la tâche répétée composée de la Figure 46 dans une tâche composée contenant une sous-tâche répétée, selon les règles suivantes :

DÉPLACEMENT DES CONNECTEURS. Les connecteurs connectés à la répétition, selon leur type, vont se retrouver soit à l'extérieur de la tâche générée (les connecteurs normaux et les reshapes), soit à

l'intérieur (les tilers, les dépendances interrépétition et les liens par défaut).

CHANGEMENT DES TAILLES DES PORTS. Suite au déplacement des connecteurs, les tailles des ports sur les bords de la tâche générée auront des tailles différentes :
- pour un tiler, la taille du tableau ($\mathbf{p_{in}}$) sur les figures) ;
- pour un lien par défaut, la taille n'est pas changée ($\mathbf{p_d}$) sur les figures) ;
- pour les connecteurs normaux et les reshapes, la taille du l'espace de répétition multipliée par la taille du port de la tâche répétée ($\mathbf{s_R}, \mathbf{p_o}$) sur les figures) ;

TAILLES RÉELLES DES TABLEAUX. Le changement des tailles selon la règle précédente nous indique que, dans une décomposition répétitive-composée, la taille réelle des tableaux des sous-tâches répétées dépend du type de connecteur qui est connecté au port ; pour les connecteurs normaux et reshapes, la taille réelle des ports est donnée par la concaténation des shapes de l'espace de répétition et du port. *La spécification doit être fait en respectant la taille réelle des ports.* Sur la Figure 46, la reshape ρ est entre un port de taille $(\mathbf{s_R}, \mathbf{p_o})$ et un port de taille $(\mathbf{p_r})$ et donc le tiler source θ_s contient des matrices d'ajustage et de pavage et le vecteur d'origine avec des dimensions qui correspondent à la taille du port réel.

GÉNÉRATION DES CONNECTEURS Le déplacement des tilers et des liens par défaut se fait avec la génération des connecteurs dans la tâche composée ;

GÉNÉRATION DU TILER IDENTITÉ. Les connecteurs normaux et les reshapes restent au niveau de la tâche composée, avec la génération d'un tiler « identité »[11] dans la répétition générée – un tiler qui place $(\mathbf{s_R})$ motifs de taille $(\mathbf{p_o})$ dans un tableau de taille $(\mathbf{s_R}, \mathbf{p_o})$,

[11] Les matrices d'ajustage et de pavage ont une forme pseudo-identité.

$$\theta_{id} = \begin{cases} \mathbf{o_{id}} = \mathbf{0} \\ F_{id} = \begin{pmatrix} 0^{dim(\mathbf{s_R}) \times dim(\mathbf{p_o})} \\ I^{dim(\mathbf{p_o})} \end{pmatrix} \\ P_{id} = \begin{pmatrix} I^{dim(\mathbf{s_R})} \\ 0^{dim(\mathbf{p_o}) \times dim(\mathbf{s_R})} \end{pmatrix} \end{cases}, \tag{3.2}$$

où I^n est une matrice carrée identité de taille n et $0^{n \times m}$ est une matrice nulle de taille $n \times m$.

Exemple. Le tiler identité qui place des motifs deux dimensionnels dans un tableau avec cinq dimensions :

$$\mathbf{o_{id}} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, F_{id} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, P_{id} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

3.7 MODÉLISATION DES ARCHITECTURES

La modélisation conjointe de l'application et de l'architecture d'un système permet d'avoir une vison d'ensemble du système et facilite

l'exploration de l'espace de modélisation et des optimisations au niveau de la spécification.

3.7.1 *État de l'art*

L'architecture d'un système est l'étape initiale dans la conception qui aide à déterminer les paramètres globaux du système et des contraintes liées à la fonctionnalité, la maintenabilité, la gestion des ressources, performances, etc. Usuellement, dans des langages tels que les ADLs, *Architecture Description Langages* – Langages de description des architectures –, des systèmes complexes d'architectures sont modélisés suivant un modèle orienté composants, où l'interaction entre les modules se fait par une *infrastructure* [54] contenant normalement des *composants*, des *ports*, des *connecteurs*, etc.

UML propose une infrastructure orientée composants et une manière facile de faire des extensions et des spécialisations, à travers les profils UML. Plusieurs profils UML ont été développés pour la conception des systèmes embarqués temps réels et plusieurs d'entre eux ont été standardisés. Deux catégories majeurs peuvent être identifiées :

A. les profils orientés vers des implémentations de nivaux bas qui s'intéressent à la réalisation des circuits, telles que UML for SoC [77] ou UML for SystemC [82] ;

B. les profils qui visent la modélisation des systèmes dans une perspective fonctionnelle. Dans cette catégorie on peut citer SPT (Schedulability, Performance and Time) [76], ¨et le premier standard proposé par OMG pour l'ingénierie système, SysML (System Modeling Language) [74].

SysML est un standard OMG qui fournit des mécanismes génériques pour la description de systèmes complexes. SysML n'apporte cependant pas une attention particulière aux systèmes embarqués, comme le fait le profil pour la modélisation et l'analyse des systèmes embarqués et temps réel MARTE. Même si SysML est utilisé par la communauté pour la conception des SoC, le standard n'était pas créé spécialement pour le domaine des systèmes embarqués et un nombre de propriétés non fonctionnelles telles que des contraintes de temps, de latence et de débit qui sont cruciales pour le design de systèmes temps réels sont absentes du profil. Ce n'est pas le cas du profil UML MARTE, qui fournit entre autres des concepts précis pour la description des détails d'architectures matérielles dans les SoCs. Dans le cas de similitudes entre des concepts du profil SysML avec des concepts du profil MARTE, MARTE réutilise les stéréotypes définis par SysML ou définit des éléments qui sont conceptuellement et terminologiquement alignés sur SysML. MARTE raffine les concepts de temps proposés dans le profil SPT (Scheduling, Performance and Time) [73] ce qui permet la modélisation de temps logique, discret et continu. Les modèles conformes au profil MARTE peuvent être annotés par des propriétés non fonctionnelles dans le but d'exprimer des contraintes auxquelles le système est soumis (contraintes de temps, de consommation d'énergie, etc.)

Plusieurs projets s'intéressent à la comodélisation Software/Hardware pour MPSoC mais seulement une partie d'entre eux intègre une méthodologie IDM ou utilisent des spécifications propres non stan-

dards. On peut mentionner ROSES [90], MILAN [64], MOPCOM [2] ou MARTES [42].

Dans cette thèse nous nous intéressons plutôt aux aspects de la modélisation répétitive dans le contexte de la co-modélisation MPSoC. Par l'utilisation conjointe des paquetages HRM et RSM de MARTE, nous pouvons modéliser des architectures répétitives sous une forme compacte et cela facilite le placement uniforme de l'application sur ces architectures.

3.7.2 *Modéliser des architectures répétées*

GASPARD2 est un environnement de comodélisation qui permet aussi la modélisation des architectures matérielles et du placement des applications sur ces architectures. En utilisant les mêmes concepts de RSM (répétitions, tilers, shapes, etc.), des architectures répétitives peuvent être modélisées sous une forme compacte.

L'environnement MMALPHA qui implémente le langage fonctionnel ALPHA, discuté dans la sous-section 1.4.1 permet la synthèse des architectures régulières (et parallèles) à partir la description par des équations récurrentes. Mozipo *et al.* présentent dans [66] la génération des circuits VLSI (des tableaux systoliques) pour le filtrage Kalman. En utilisant l'environnement MMALPHA pour résoudre le problème de programmation linéaire entre les variables, les auteurs arrivent à générer une architecture systolique où le parallélisme est maximal. Bednara et Teich dans [8] visent le placement automatique des programmes contenant des nids de boucles sur des architectures hardware reconfigurables en utilisant l'outil de design PARO. Les deux propositions visent la génération automatique du code VHDL synthétisable pour l'implémentation en FPGA des tableaux de processeurs systoliques.

L'environnement GASPARD2 est aussi capable de générer du code VHDL pour la synthèse de FPGA, mais on ne peut pas prétendre d'attendre les performances de MMALPHA, même si les modules VHDL générés reflètent la spécification de haut niveau et des transformations de haut niveau peuvent être utilisées pour améliorer les performances.

La modélisation répétitive en MARTE des architecture vise plutôt l'expression compacte des architectures répétitives qui facilite énormément la spécification et permet une exploration des distributions des calculs sur les unités de calculs parallèles. Une modélisation compacte peut être expansée par une transformation de mise à plat qui replace les répétitions avec des copies réelles et les liens de topologie avec des liens de connexion élémentaires.

3.7.3 *Concepts répétitifs dans des structures non applicatives*

La définition statique ARRAY-OL introduit des contraintes dans la spécification des applications en MARTE pour GASPARD2. Ces contraintes concernent surtout les liens topologiques et sont là pour garantir la définition statique de la modélisation. Les règles de spécification assurent l'assignation unique et la production entière de tous les éléments des tableaux.

Dans les modélisations non applicatives, les concepts répétitifs ont une signification légèrement différente. Les liens topologiques n'expriment plus des dépendances de données, mais des liens physiques entre des unités matérielles et les règles de la définition statique ARRAY-OL

ne s'appliquent plus. Par la suite nous allons voir quels sont les changements le plus importants dans la modélisation et leur signification.

Ports « InOut »

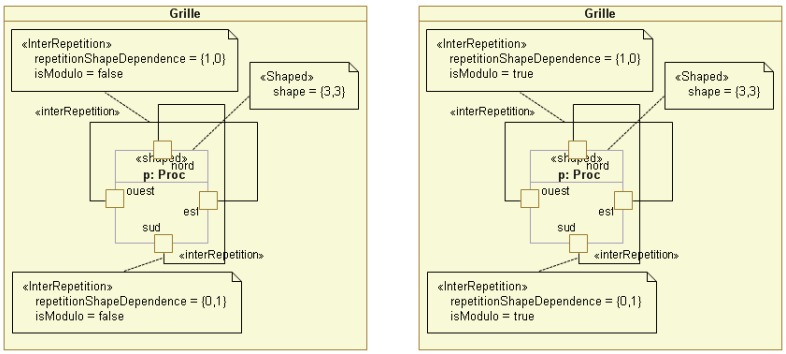
MARTE permet la spécification des ports de flot bidirectionnels – entrée-sortie – pour exprimer dans l'architecture des canaux de communication avec double sens. L'étiquette de direction du stéréotype *FlowPort* permet la spécification de la direction d'un port dans un modèle MARTE, *DirectionKind*::*in*, *DirectionKind*::*out*, ou *DirectionKind*::*inout*.

Tilers non tuilés parfaitement

Dans la spécification des applications Array-OL, les tilers de sortie (ceux qui réarrangent les motifs produits par la répétition dans les tableaux de sortie) doivent tuiler parfaitement les tableaux de sortie ; tous les éléments des tableaux sont produits et les tuiles ne se chevauchent pas. Dans la modélisation des architectures, cette contrainte ne s'applique plus et les structures répétitives sont plus flexibles. La même remarque s'applique aussi aux tilers des reshapes.

Dépendances interrépétitions cycliques

Les dépendances interrépétitions dans la spécification de l'application sont obligatoirement non cycliques. Une éventuelle présence d'une dépendance cyclique introduirait des dépendances de données cycliques et donc des points de blocage dans l'exécution. Néanmoins, dans la modélisation de l'architecture, ce type de dépendance est permis et est très utile pour exprimer des constructions telles que des anneaux de processeurs interconnectés. Un exemple d'une telle construction est montré sur la Figure 48, qui spécifie des structures de la Figure 49, où les répétitions sont mises à plat.

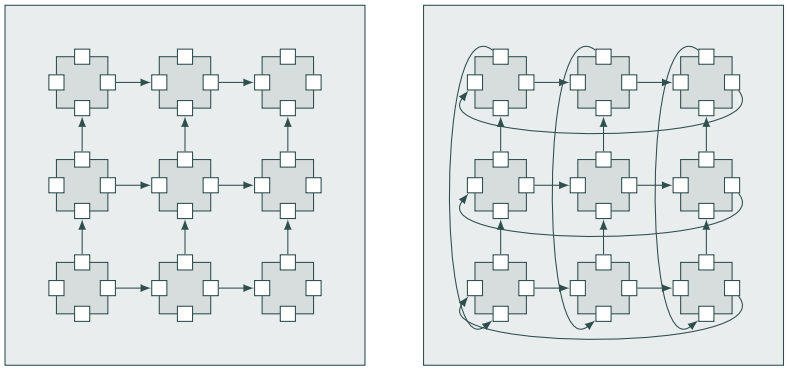


(a) Dépendances non-cycliques (b) Dépendances cycliques

Fig. 48: Grille de processeurs interconnectée - modélisation MARTE RSM

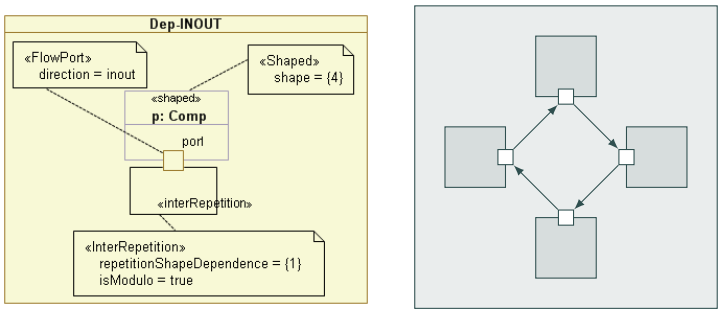
Interrépétition sur le même port

Les ports de la modélisation de l'architecture peuvent être bidirectionnels (*inout*) et ainsi une dépendance inter-répétition, qui par définition est entre un port de sortie et un port d'entrée, peut être définie sur le même port bidirectionnel. Figure 50 montre une telle construction.



(a) Dépendances non-cycliques (b) Dépendances cycliques

FIG. 49: Grille de processeurs inter-connectées



(a) Modèle UML (b) Topologie mise-à-plat

FIG. 50: Dépendance sur le même port

Reshape sur le même port

La décomposition répétitive permet la spécification compacte des architectures régulières. Des constructions plus complexes peuvent être modélisées en faisant usage de la hiérarchie. Pour spécifier des liens moins réguliers, la construction de reshape peut être employée. Elle est capable d'exprimer des liens :

- entre seules parties des éléments des deux ensembles de ports connectés ;
- de plusieurs éléments vers un ou de un vers plusieurs ;
- réarrangeant des éléments dans des tableaux multidimensionnels ;
- etc.

Le reshape peut être utilisé sur le même port, cas où il exprime des liens entre les éléments du même tableau. Cela permettrait par exemple la spécification des inter-connexions moins réguliers que les dépendances interrépétition.

Exemple. Dans l'Appendice B, la modélisation d'une architecture NoC est montrée. La décomposition hiérarchique et les concepts répétitifs de MARTE RSM sont utilisés pour exprimer sur une forme compacte une architecture où la régularité n'est pas directement exprimable par les concepts répétitifs de base.

Après avoir discuté des aspects de la modélisation répétitive des applications et des architectures en MARTE, nous allons nous pencher sur le placement des applications sur les architectures.

3.8 PLACEMENT RÉPÉTITIF

Le profil MARTE permet la comodélisation de l'application et de la plateforme d'exécution. Les deux sont spécifiés séparément au début, suivi par le placement de l'application sur l'architecture matérielle pour exprimer le système complet.

Le placement efficace des calculs sur les unités d'exécution est un aspect essentiel de la comodélisation pour des systèmes embarqués. Celui a une influence forte sur les performances et un rôle important sur le plan de la consomption en énergie. Pour permettre un placement optimal, plusieurs possibilités devront être testées et comparées.

Le concept de placement est présent en MARTE (paquetage *Alloc*) et dans RSM une extension répétitive est proposée.

3.8.1 *Allocation*

Une allocation MARTE est une association entre un application MARTE et une plateforme d'exécution MARTE. L'ensemble de toutes les allocations du modèle définit le placement complet.

Le concept principal est représenté par le stéréotype « Allocate », utilisé pour spécifies des associations entre les éléments du modèle de l'application et des éléments du modèle de l'architecture.

L'allocation peut représenter soit un placement spatial soit un placement temporel. Le placement spatial est le placement des calculs sur des ressources de calcul, des données sur des mémoires à des plages spécifiques d'adresses et des dépendances entre des ressources logicielles et des ressources de communication. Le placement temporel représente l'ordonnancement d'une série d'éléments placés sur la même ressource matérielle. Par exemple, plusieurs tâches peuvent être placées sur le même processeur ou une plage d'adresses peut être utilisée aux instances différentes de temps pour contenir des données différentes. Une allocation est valide si les deux types d'allocation, spatiale et temporelle, sont consistants.

3.8.2 *Distribution*

Nous avons vu comment les constructions de MARTE RSM permettent en même temps la spécification répétitive du parallélisme disponible dans l'application et la spécification compacte des architectures répétées. Dans les deux cas, les répétitions sont exprimées avec l'aide des espaces multidimensionnels. Le placement de telles applications sur de telles architectures peut se faire d'une manière distribuée, en utilisant un concept qui englobe le reshape et l'allocation, la *distribution* entre des éléments répétés dans un modèle d'application et des éléments répétés dans un modèle d'architecture. Le sujet est abordé par Boulet *et al.* dans [15].

La Figure 51 montre le concept de distribution dans le profil MARTE.

Ce concept peut être employé pour exprimer des distributions uniformes :

A. des répétitions sur des unités de calculs parallèles ;

B. des tableaux sur des mémoires distribués.

Quelques observations sur telles constructions sont nécessaires :

– chaque répétition doit être placée sur une unique unité de calcul ;

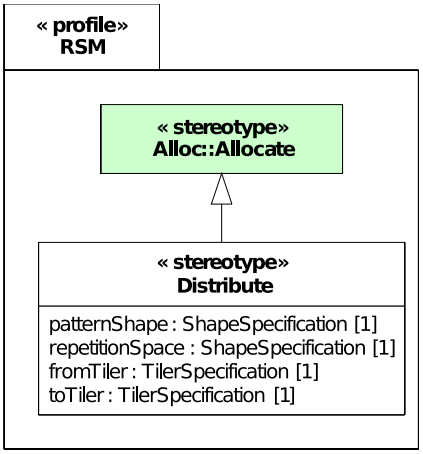


FIG. 51: Distribute en MARTE RSM

– chaque élément du tableau doit être placé sur une unique mémoire.

Ces observations se traduisent sur la contrainte que le tiler source doit paver exactement l'espace multidimensionnel source.

Exemple. La Figure 52 montre la distribution par colonne d'une répétition bidimensionnelle sur 4 processeurs.

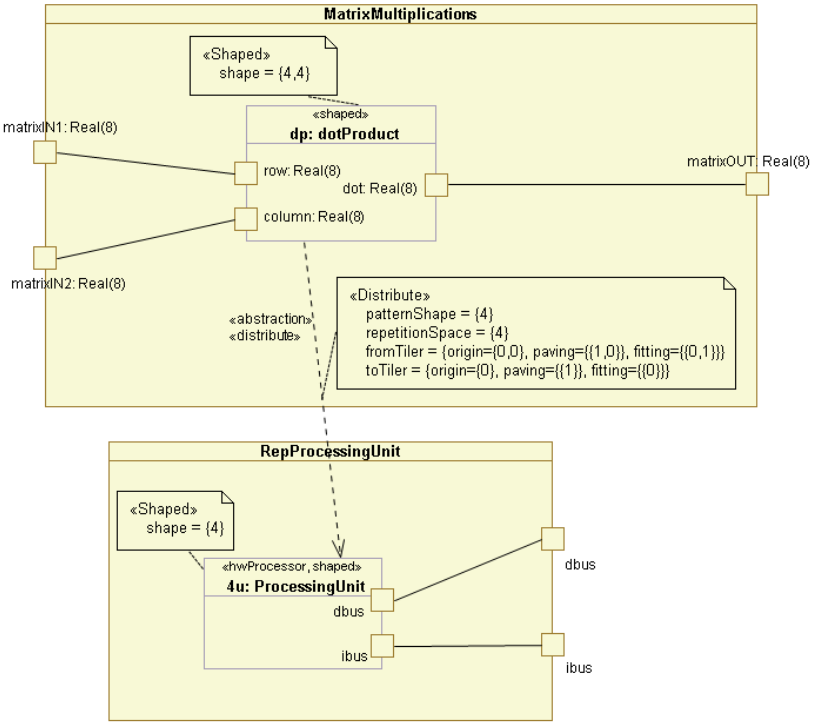


FIG. 52: Distribution d'une répétition de 4×4 sur 4 processeurs

Les deux répétitions et le placement sont montrés sur la Figure 53.

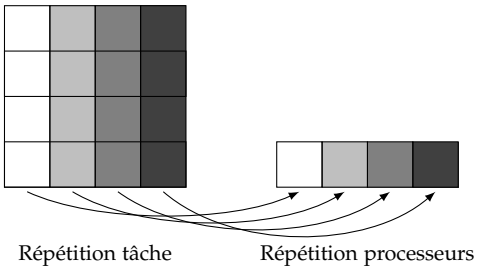


FIG. 53: Placement des répétitions de la distribution

3.9 CONCLUSIONS

Dans ce chapitre nous avons vu comment le profil UML de MARTE peut être utilisé pour la comodélisation des applications de traitement intensif de signal sur SoC. Nous nous sommes intéressés surtout à l'aspect répétitif de la modélisation en MARTE RSM, qui permet la modélisation du parallélisme dans l'application, des architectures répétées sous une forme compacte et une allocation distribuée de l'application sur l'architecture.

Toutes les concepts de topologie répétitive d'ARRAY-OL se retrouvent dans la modélisation MARTE et une modélisation en MARTE est équivalente ou peut être réduite au modèle de calcul ARRAY-OL et donc la modélisation d'une application conserve la propriété de la définition statique qui offre la possibilité de calcul statique d'un ordonnancement.

Nous avons vu aussi comment les concepts de topologie répétitive ont une sémantique et une signification légèrement différente dans la modélisation de l'architecture. La représentation compacte d'une architecture répétée facilite la modélisation – surtout quand il s'agit des architectures complexes avec un nombre important d'unités interconnectées – et permet l'exploration pour un placement performant.

Dans les chapitres suivants, nous allons nous intéresser aux aspects liés à l'exécution et aux optimisations de haut niveau pour adapter la spécification à l'exécution.

Deuxième partie

DE LA SPÉCIFICATION VERS L'EXÉCUTION

4 OPTIMISATIONS SOURCE À SOURCE

Dans les chapitres précédents, nous avons discuté de la comodélisation répétitive pour des applications de traitement intensif de signal dans le contexte des systèmes embarqués. La qualité de la spécification est essentielle pour arriver à générer du code performant. Un modèle de spécification exprime la fonctionnalité d'une application, en exprimant toutes les dépendances de données, mais une telle spécification ne dit rien sur la façon dont exécuter l'application.

Tous les ordonnancements d'exécution qui respectent les dépendances de données sont corrects. Mais lequel est l'ordonnancement optimal ? Ce problème est complexe et dans l'équipe nous avons choisi d'utiliser des optimisations à plusieurs niveaux d'abstraction. Des optimisations de haut niveau visent l'adaptation de la spécification à l'exécution, en réduisant les tableaux intermédiaires et changeant la granularité de l'application. Un passage direct à un modèle d'exécution est fait, ayant comme avantages que la structure de l'application se retrouve à l'exécution et que le concepteur a un degré de contrôle sur le code généré et exécuté, mais on peut perdre en performances.

Avant de s'intéresser aux optimisations de haut niveau dans le contexte de la spécification en MARTE, nous allons discuter du passage à l'exécution et des techniques d'optimisation basées sur des transformations de boucles, qui partagent des éléments en commun avec les transformations Array-OL, mais à un autre niveau d'abstraction.

La section 4.1 analyse l'exécution d'une spécification Array-OL et dans section 4.2 nous faisons état des techniques d'optimisations de code par des transformations de boucles.

4.1 ÉTUDE DE L'EXÉCUTION

Une *spécification* MARTE RSM exprime le maximum de parallélisme, en utilisant une décomposition hiérarchique et répétitive. Elle décrit les dépendances de données entre les éléments des tableaux et, comme conséquence directe, un ordre partiel strict entre les appels des tâches. Une spécification valide dans le sens de Gaspard2 doit être définie statiquement et ainsi admettre un ordre partiel strict. Effectivement, quelconque ordonnancement qui respecte cet ordre calculera les mêmes valeurs de sortie à partir des mêmes entrées.

Les règles de construction qui assurent la validité d'une spécification sont rappelées dans la section 2.4.

Un *modèle d'exécution* représente l'abstraction de l'exécution et il doit respecter l'ordre partiel strict défini par la spécification statique. En exprimant l'ordre minimal d'exécution, de nombreuses décisions peuvent et doivent êtres prises au moment du placement d'une spécification MARTE sur une plateforme d'exécution :

A. comment associer les différentes répétitions en temps ou espace ?
B. comment placer les tableaux dans la mémoire ?
C. comment ordonnancer les tâches parallèles sur la même unité de calcul ?
D. comment ordonnancer les communications entre les unités de calculs ?

La séparation entre l'association espace-temps et la spécification fonctionnelle est un des points forts de la modélisation en MARTE. Cela permet la construction des bibliothèques fonctionnelles pour la réutilisation et d'effectuer des explorations architecturales avec les moins de restrictions possibles.

Les premiers travaux sur l'exécution d'une application conforme au modèle de calcul Array-OL sans hiérarchie ont été réalisés par Ancourt *et al.* dans le cadre d'une collaboration avec TUS[1] [4] en 1997. Ces travaux étudient l'ordonnancement et le placement automatique d'une application sur une architecture SPMD.

[1] TUS = Thales Underwater Systems.

Ils proposent pour cela d'utiliser un modèle concurrent de programmation logique par contrainte (CCLP) [31, 43]. Les contraintes sont établies à partir d'une formalisation de l'architecture, de l'application et du placement. Il est ainsi possible de tenir compte d'un grand nombre de paramètres au cours de la formalisation : le nombre de processeurs, la taille de la mémoire, la latence maximale entre les entrées et les sorties, etc. Les contraintes sont passées à un outil de résolution qui utilise un ensemble d'heuristiques pour obtenir : un ordonnancement des calculs, un placement de ces calculs sur les processeurs et une gestion optimisée de la mémoire.

Au final, cette méthode est capable de générer automatiquement un ordonnancement séquentiel, pipeline ou data-parallèle, mais sans toutefois garantir que cet ordonnancement soit optimum. Afin d'améliorer la qualité des résultats, Ancourt *et al.* proposent de restructurer l'application pour obtenir un plus haut degré de parallélisme et pour améliorer la localité des données. Les outils de refactoring décrits dans cette thèse pourraient donc être utilisés en aval de ces calculs d'ordonnancement.

4.1.1 *Modèle d'exécution immédiat*

Nous avons choisi pour le passage d'une spécification MARTE vers un modèle d'exécution de le faire le plus directement possible : *le modèle d'exécution devrait refléter la spécification.* Ce choix a l'avantage de garder des similitudes avec la structure de la spécification (parallélisme, granularité) dans l'exécution, au moins jusqu'à la génération de code quand des optimisations de bas-niveau entrent en jeu.

Exemple. Un exemple représentatif est la génération de code VHDL utilisé par la suite pour la synthèse des circuits FPGA[2], où la structure de la spécification se retrouve au niveau des modules VHDL et de la disposition FPGA.

[2] Disponible dans l'environnement Gaspard2.

Le choix de ce modèle d'exécution a ses désavantages naturellement. Un autre choix pourrait être un modèle d'exécution dérivé des dépendances exactes de la spécification, suite aux règles formelles qui définissent les dépendances globales au niveau des éléments des tableaux. Une telle direction risque de réduire le contrôle du modeleur sur l'exécution, mais les gaines en performances pourront être importantes.

Ce passage direct se traduit par la contrainte qui dit qu'une tâche ne peut commencer son exécution que quand les tableaux d'entrées sont entièrement disponibles. Cette contrainte introduit un problème majeur, ce que nous appelons des « barrières de synchronisation » entre les composants. Une telle barrière est créée par les dépendances de données et une illustration représentative est la présence d'un tableau intermédiaire avec une dimension infinie qui va causer un blocage dans l'exécution dans ce point. Comme solution, certaines répétitions peuvent être transformées en flots, l'exécution des répétitions est séquentialisée (ou pipelinée) et les motifs sont consommés et produits comme un flot de jetons (chaque jeton transportant un motif).

Exemple de blocage : une tâche attend qu'une autre finît de produire un tableau infini.

4.1.2 *Refactoring pour l'adaptation de la spécification vers l'exécution*

Le refactoring de l'application pour l'adaptation de la spécification au passage à l'exécution joue un rôle important dans le but d'augmenter les performances du code généré.

En utilisant des transformations de haut niveau data-parallèles, le refactoring permet le changement de la granularité des répétitions dans la description répétitive, la réduction des tableau intermédiaires et l'augmentation de la localité des données. Il facilite l'exploration de l'espace de conception pour prendre en compte les contraintes de ressources et de l'environnement.

Ce sont des outils d'optimisation au niveau haut de la spécification et ils se traduisent par des changements dans la spécification ; l'utilisation d'autres optimisations à un niveau bas est toujours possible. Par l'utilisation d'un modèle d'exécution immédiat, les optimisations de haut niveau peuvent être guidées par la simulation du système au niveau de l'implémentation[3].

Les transformations data-parallèles ressemblent aux transformations de boucles usuelles, mais au niveau de la spécification. Les techniques d'optimisation basées sur les transformations de boucles sont résumées brièvement par la suite pour identifier des connexions avec les transformations Array-OL.

[3] Dans l'environnement Gaspard2, la simulation peut se faire en SystemC au niveau TLM.

4.2 OPTIMISATIONS DU CODE

Le décalage entre les performances de pointe clamées des constructeurs et celles achevées avec le code réel a radicalement augmenté dernièrement, principalement à cause de l'augmentation brutale de la complexité des processeurs qui a entraîné une dégradation importante de l'efficacité du code généré par les compilateurs.

Les trois directions principales pour améliorer les performances sont :

1. accroître le parallélisme au niveau des instructions en même temps que de multiplier les mécanismes permettant l'exécution simultanément des instructions et réduisant le plus possible leur latence ;

2. faire évoluer les mécanismes spéculatifs permettant la prédiction du comportement local des programmes ;
3. implémenter des hiérarchies mémoires complexes pour exploiter le mieux la localité des données, spatiale ou temporale.

Pour toutes ces directions, les techniques de transformations de code « source à source » ont un rôle déterminant. La plupart de ces techniques sont représentées par des transformations appliquées aux nids de boucles « for » qui peuvent être utilisés dans deux sens :

A. augmenter le parallélisme des instructions ;
B. améliorer la régularité et la localité des accès aux données et, en plus, l'élimination des tampons au niveau système dans le code.

Ces transformations peuvent être utilisées surtout dans le cas du code extrêmement régulier qui contient du code de traitement des données fortement uniforme. Cela favorise le domaine d'application du traitement de signal intensif orienté flot de données.

La plupart des transformations qui visent l'optimisation des programmes pour des monoprocesseurs réduisent le nombre d'instructions exécutées par l'analyse des quantités scalaires et des techniques orientées flot de données. En revanche, les optimisations pour des processeurs haute-performance, vectoriels ou parallèles visent à maximiser le parallélisme et la localité mémoire, avec des transformations qui se basent sur le repérage des caractéristiques des tableaux en utilisant des analyses des dépendances de boucles.

4.2.1 *Transformations de boucles*

Une technique importante au niveau système, la technique des transformations des boucles « for » vise l'augmentation de la régularité et de la localité des accès aux données en permettant la suppression (ou la réduction) des tampons dans le code. Ainsi, elle permet la réduction des besoins en taille mémoire globale et les latences des accès mémoire. Il est vital pour la taille, la consommation d'énergie et les performances des systèmes embarqués. Une régularité et une localité améliorée augmentent le ratio de réutilisation des emplacements mémoire, puisque des zones mémoire peuvent être utilisées pour des éléments de données avec des durées de vie non chevauchantes. Cela, à son tour, réduit les besoins en taille mémoire globale. L'amélioration de la régularité des accès aux données peut aussi augmenter le degré de parallélisme d'une application.

Nous n'avons pas l'intention de faire une liste complète des transformations de code existantes, le sujet et extrêmement complexe et a été au cœur de nombreuses études, tels que Zima et Chapman en [94], Darte *et al.* en [23], Kennedy et Allen en [51] ou Wolfe en [92]. Bacon *et al.* donnent dans [7] un vue d'ensemble sur les techniques de restructuration autour des boucles dans un langage impératif.

Les méthodes de transformation de boucles peuvent être classées selon différents critères et leur rôle peut changer en fonction de l'architecture ciblée et des objectifs ; augmenter le parallélisme et la réduction de la taille mémoire peuvent être des objectifs divergents dans certains cas.

Dans le cadre du projet Ter@ops[4] du pôle de compétitivité System@tic, un glossaire des transformations de code a été établi, relativement aux différents outils participants au projet et leurs techniques

[4] http://teraops-emb.ief.u-psud.fr/

d'optimisations[5]. Plusieurs types de transformations de code ont été identifiés :

[5] http://www.ief.u-psud.fr/~tadonki/projects/teraops/glossaire/

MODIFICATION DE L'ALLOCATION MÉMOIRE : renommage de scalaires, expansion de scalaires/tableaux, remplacement d'un tableau global par un scalaire privé, etc. ;

TRANSFORMATIONS DE BOUCLES : découpage de l'espace d'itérations, déroulage (partiel ou complet) de boucle, fusion de boucles, fission de boucles, découpage de l'espace d'itérations et beaucoup d'autres ;

TRANSFORMATIONS INTER-PROCÉDURALES : déplacement des boucles entre l'extérieur et l'intérieur d'une procédure (ou l'inverse), le remplacement d'un appel de procédure par son corps, etc. ;

ÉLIMINATION DU CODE INUTILE.

Ces transformations de code, même si elles sont la plupart de temps utilisées autour les boucles, sont liées aussi aux concepts du code impératif : scalaires, procédures, itérations de boucles. Dans le contexte de la spécification répétitive en MARTE, les répétitions peuvent être vus comme des nids de boucles avec une structure à part, sans des variables locales, scalaires, conflits de données et appels aux procédures. Dans ce sens, nous allons restreindre notre intérêt aux transformations de nids boucles parfaits.

Nous allons montrer une sélection des transformations importantes dans le contexte de cette thèse :

DÉPLACEMENT DU CODE qui change l'ordre d'exécution entre deux boucles dans le code, sans changer les boucles elles-mêmes. La transformation peut être utilisée pour augmenter la localité des données ou comme support pour des autres transformations.

FUSION DE BOUCLES qui groupe plusieurs boucles successives dans une seule. Elle peut être utilisée pour réduire les tableaux intermédiaires et ainsi les besoins en taille mémoire.

SCISSION DE BOUCLES qui effectue l'opération inverse de la fusion et qui essaye de simplifier une boucle ou d'éliminer des dépendances en cassant la boucle en multiples boucles avec le même corps de boucle, mais itère sur des portions continues différentes de l'espace initial d'itération. Elle peut être utilisée aussi pour séparer les instructions parallèles du reste.

TILING (OU PARTAGE) DE BOUCLES augmente le niveau de nidification d'une boucle. L'effet est que l'espace d'itération est divisé en blocs qui sont traités en séquentiel. Le partage de l'espace d'itération entraîne le découpage des grands tableaux en blocs plus petits et ainsi on peut rentrer les éléments accédés dans le cache, améliorer la réutilisation du cache et réduire les besoins en taille de cache.

DÉROULAGE DE BOUCLES réduit le nombre d'itérations en dédoublant les mêmes instructions dans le corps de la boucle, dans le but de limiter l'évaluation de la condition de la boucle et des sauts, qui pénalisent la performance en affectant le pipeline des instructions.

PIPELINE DE BOUCLES (ALIGNEMENT OU PLIAGE) décale certaines instructions d'une ou plusieurs itérations dans la boucle. Cette

transformation est utilisée pour améliorer la localité des données et peut être aussi utilisée comme transformation support.

EFFONDREMENT DE BOUCLES est l'inverse du tiling. Les deux peuvent être utilisées comme outils de manipulation du concept de blocs de données dans l'application, essentiel pour la localité des données et le parallélisme.

Toutes ces transformations et encore beaucoup plus[6] sont utilisées ensemble pour attendre les meilleures performances. Le choix des transformations à appliquer et l'ordre sont essentiels et peut dépendre du but final de l'optimisation.

[6] Les transformations listées sont qu'un sous-ensemble des transformations de boucles existantes.

4.2.2 *Techniques d'optimisation des boucles*

Typiquement, une optimisation du code au niveau du compilateur consiste en trois pas :

1. choisir une partie de code à optimiser et l'enchaînement de transformations à appliquer ;
2. vérifier la conformité de l'optimisation ;
3. et en dernier, appliquer les transformations.

La première étape est la plus laborieuse et parce que l'analyse est coûteuse, des obstacles d'ingénierie très souvent mettent des contraintes sur les stratégies d'optimisation disponible au compilateur. Comme l'architecture du processeur est de plus en plus complexe, le nombre de dimensions dans lequel l'optimisation est possible augmente, ce qui rend le processus de décision extrêmement complexe.

La conformité d'une transformation et un aspect extrêmement important dans le contexte des optimisations de code. Nous devrons nous assurer que l'application d'une transformation (ou d'une série de transformations) ne modifie pas le fonctionnement du code. Les restrictions sont usuellement exprimées sous la forme de dépendances (de données ou de contrôle). La partie complexe des optimisations est l'identification des transformations correctes (qui ne changent pas les dépendances dans l'application) et le choix de la chaîne optimale de transformations pour atteindre nos objectifs.

Un algorithme parfait d'optimisation celui que l'un proposé par Kennedy et McKinley pour la réutilisation maximale par des fusions des boucles est prouvé extrêmement coûteux en termes de complexité, temps et ressources [52] – problème *NP*-complet – et cela a conduit à l'utilisation extensive des heuristiques. La plupart des compilateurs intègrent une série de décisions heuristiques relatives aux ordonnancements des transformations susceptibles de fonctionner avec des bons résultats sur la/les machine(s) cible. Les optimisations peuvent prendre place à des phases distinctes de la compilation, il n'y a pas d'organisation définitive. Des architectures dissemblables dictent des designs différents et les opinions sur le meilleur ordre font polémique. Bacon *et al.* présentent un tel design dans [7] pour donner une idée sur comment les transformations s'assemblent.

La complexité des algorithmes d'optimisation est une raison pour laquelle beaucoup de compilateurs (pour ne pas dire la plupart) utilisent encore des heuristiques. Cela implique l'utilisation de la même chaîne de transformations, celle qui statiquement démontre les meilleurs résultats dans la plupart des cas. Des compilateurs plus complexes disposent

de plusieurs chaînes de transformations d'où choisir, selon les caractéristiques de l'application.

La compilation itérative [39, 53] est une approche répandue pour l'optimisation des programmes pour des objectifs variés sur des architectures d'une complexité sans cesse croissante, lorsque les compilateurs traditionnels ne parviennent pas à offrir la meilleure performance possible. Car construire des modèles détaillés de coût statique pour les architectures modernes et dynamiques n'est pas plus possible, la compilation itérative s'appuyer sur le mécanisme où des transformations successives ont été appliquées à un programme et leur une valeur déterminée par l'exécution effective du code qui en résulte. Un grand nombre de différentes versions du programme sont générées et exécutées, avec la plus rapide version sélectionnée. Une telle approche est décidable et, étant donné suffisamment de temps, sera trouvé le meilleur programme. L'inconvénient évident est que le temps de compilation de façon spectaculaire augmentations.

La complexité du sujet a suscité la nécessité de concevoir des moyens de représentation du problème (contraintes, transformations, fonction de coût) utilisant un formalisme plus efficace qui pourrait faciliter la manipulation des concepts tels que la conformité, les dépendances de données et de contrôle, les objectifs d'optimisation. Certains ont approché le thème en utilisant l'*algèbre linéaire* (Feautrier en [36]), *abstraction polyédrale* (Girbal en [41]), algorithmes de la *théorie des graphes* ou la *programmation linéaire entière* (Fraboulet en [38]). Toutes les approches ont en commun, à part du fait d'obtenir une solution presque optimale, la forte complexité.

L'introduction du formalisme est une prémisse importante pour la partie décisionnelle de l'optimisation. Des algorithmes corrects et efficaces ont besoins d'être conçus autour de tels formalismes.

4.2.3 *Fonctionnement de l'optimisation des boucles*

Pour mieux comprendre l'utilisation des transformations de boucles, un exemple pourrait être plus approprié. L'exemple classique qui suit présente la fonctionnalité de la transformation de *fusion de boucles*.

Exemple. Ici, un tableau est écrit dans une boucle et lu dans une autre. Entre les deux boucles, les tableaux doivent être gardés en mémoire.

```
1: for i = 1 to n do
2:     A[i] = expr1
3: end for
4: for i = 1 to n do
5:     expr2 = f(A[i])
6: end for
```

La technique de fusion des boucles permet le groupement de plusieurs boucles dans une seule. Le résultat peut être observé ci-dessous :

```
1: for i = 1 to n do
2:     A[i] = expr1
3:     expr2 = f(A[i])
4: end for
```

La fusion n'a pas d'impact sur la consomption en taille mémoire. C'est parce que la fusion des boucles ne travaille pas toute seule. Elle est utilisée en collaboration avec des autres transformations, le *replacement*

scalaire dans ce cas. Cette technique peut supprimer des tableaux entiers de la mémoire, en les remplaçant avec des scalaires, comme montré ci-dessous.

Exemple. Ce remplacement et correct seulement si les valeurs de tableau **A** produites par la première boucle ne sont pas utilisées ailleurs.

```
1: for i = 1 to n do
2:     a = expr1
3:     expr2 = f(a)
4: end for
```

Observation. Le remplacement du tableau **A** avec un scalaire réduit les besoins en taille mémoire, mais élimine le parallélisme de la boucle : avant le remplacement scalaire, la boucle était complètement parallèle alors qu'après elle est séquentialisée par le conflit d'écriture sur la même variable.

Les dépendances peuvent être compliquées et ne pas permettre la suppression des tableaux entiers. Dans ce cas, d'autres techniques peuvent être employées, telles que l'*optimisation de l'ordre de stockage intratableau*[7], qui sert à calculer une fenêtre de référence pour le tableau et à plier le tableau. Le fonctionnement est illustré sur l'exemple qui suit.

[7] *Intra-array storage order optimization* en anglais.

Exemple. Le code

```
1: for i = 1 to n do
2:     A[i] = expr1
3: end for
4: for i = 1 to n do
5:     expr2 = f(A[i − 1], A[i])
6: end for
```

est transformé en

```
1: for i = 1 to n do
2:     a[i%2] = expr1
3:     expr2 = f(a[i%2 − 1], a[i%2])
4: end for
```

par la fusion des boucles en collaboration avec l'optimisation de l'ordre de stockage intratableau.

Comme montrée sur les exemples précédents, la fusion des boucles peut être un outil puissant si utilisé en association avec d'autres transformations, permettant la réduction des tableaux intermédiaires et les besoins en taille mémoire. Les exemples en haut sont très simples ; pour des applications réelles, les décisions sont considérablement plus complexes, mais la manipulation d'une manière très régulière qu'on peut retrouver dans les applications orientées flots de données telles que des traitements multimédias font que ces techniques sont extrêmement efficaces.

Le problème d'optimisation basé sur des transformations de boucles pour une partie du code source peut être réduit à :

> *Trouver l'enchaînement optimal des transformations de boucles qui, appliqués sur le code d'entrée, maximiseraient la fonction de coût associé et en même temps garantirait la conformité du code de sortie.*

4.3 CONCLUSIONS

Les techniques d'optimisation de code par des transformations de boucles sont largement utilisées dans les compilateurs et leurs performances sont directement influencées par la structure du code : les nids de boucles parfaits facilitent la tâche du compilateur, pend que la présence des pointeurs rendre l'analyse des dépendances extrêmement complexe.

Une description répétitive en MARTE peut être vue comme une structure spéciale de nids de boucles avec des caractéristiques qui favorisent l'utilisation de techniques similaires aux transformations de boucles usuelles, mais à un niveau haut de la spécification.

Nous allons voir par la suite comment les structures répétées de MARTE peuvent être représentés comme nids de boucles parfaits et data-parallèles et comment des transformations data-parallèles ont été conçues autour, en utilisant un formalisme basé sur l'algèbre linéaire pour exprimer les concepts de dépendances, conformité et objectif.

5

REFACTORING EN UTILISANT DES TRANSFORMATIONS DE HAUT NIVEAU

Dans ce chapitre nous nous intéressons aux transformations de haut-niveau data-parallèles conçus autour la description répétitive. Ces transformations partagent des caractéristiques avec les transformations de boucles que nous avons vues mais qui agissent au niveau haut de la spécification multidimensionnelle.

Nous commençons dans la section 5.1 par montrer comment une description répétitive peut être traduite en un nid de boucles spécial, avant de présenter dans la section 5.2 le formalisme utilisé derrière les transformations pour la manipulation des concepts de dépendances multidimensionnelles, conformité, objectifs.

Les transformations de haut-niveau disponibles déjà existantes sont présentées brièvement dans la section 5.3.

Dans la section 5.4 nous proposons des extensions aux transformations existantes visant soit le regroupement des transformations élémentaires dans des transformation rassemblant plusieurs transformations élémentaire avec des propriétés nouvelles, soit d'étendre le domaine d'applicabilité.

La section 5.5 présente une comparaison entre les transformation de haut niveau Array-OL avec les transformations de boucles, pour identifier des relation permettant l'utilisation des résultats du domaine d'optimisations basées sur des transformations de boucles dans le contexte d'optimisations en utilisant le refactoring Array-OL.

5.1 RÉPÉTITIONS SOUS LA FORME DE BOUCLES

Une spécification MARTE RSM est conçue autour du concept de *répétition*, qui peut être vue comme l'abstraction d'un nid de boucles spécial, sous une forme visuelle data-parallèle. Les caractéristiques principales d'un tel nid de boucles sont :

- les boucles sont parfaitement imbriquées ;
- toutes les boucles varient entre zéro et une constante, avec un pas de 1 ;
- toutes les instructions se retrouvent au niveau le plus profond du nid de boucles ;
- les écritures des éléments sont en assignation unique, donc il n'y a pas de dépendances d'écriture de données, ce qui rend les boucles complètement parallèles.

L'accès au éléments des tableaux d'entrées et de sortie se fait par la construction de tiler, *cf* Équation 2.3, page 36. Pour rappeler,

$$\begin{aligned} &\forall \mathbf{r}, \mathbf{0} \leqslant \mathbf{r} < \mathbf{s}_{\text{répétition}}, \\ &\quad \forall \mathbf{i}, \mathbf{0} \leqslant \mathbf{i} < \mathbf{s}_{\text{motif}}, \\ &\quad\quad \mathbf{e}_{\mathbf{r}_i} = \mathbf{o} + (P\ F) \cdot \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s}_{\text{tableau}}, \end{aligned} \tag{5.1}$$

où

$\mathbf{e}_{\mathbf{r}_i}$	la référence dans le tableau qui corresponde a l'élément avec l'indice $\mathbf{i}$ dans le motif de la répétition $\mathbf{r}$
$\mathbf{s}_{\text{tableau}}$	la forme du tableau
$\mathbf{s}_{\text{motif}}$	la forme du motif
$\mathbf{s}_{\text{répétition}}$	la forme de l'espace de répétition
$\mathbf{o}$	l'origine du tiler
P	la matrice de pavage
F	la matrice d'ajustage.

Une répétition caractérisée par un espace de répétition $\mathbf{s}_{\text{répétition}}$ qui consume n tableaux d'entrées et produit m tableau de sortie, accès exprimés par les tilers associés qui définissent des relations de type de l'Équation 5.1 :

CONSOMPTION DES ENTRÉES : $\forall k, 0 \leqslant k < n$, le tiler d'entrée $\theta_{e_k} = (F_{e_k}, \mathbf{o}_{\mathbf{e_k}}, P_{e_k})$ exprime le placement de Motif_Opérande$_k$ avec une forme de $\mathbf{s}_{\text{motif\_opérande}_k}$ dans le Tableau_entrée$_k$ avec une forme de $\mathbf{s}_{\text{tableau\_entrée}_k}$;

PRODUCTION DES SORTIES : $\forall h, 0 \leqslant h < m$, le tiler de sortie $\theta_{s_h} = (F_{s_h}, \mathbf{o}_{\mathbf{s_h}}, P_{s_h})$ exprime le placement de Motif_Résultat$_h$ avec une forme de $\mathbf{s}_{\text{motif\_résultat}_h}$ dans le Tableau_sortie$_h$ avec une forme de $\mathbf{s}_{\text{tableau\_sortie}_h}$.

L'algorithme qui montre la traduction d'une répétition vers des nids des boucles est montré sur l'Algorithme 1.

Chaque boucle multidimensionnelle représente un nid de boucles parallèles avec la profondeur de la dimension du vecteur où chaque boucle itère une dimension du vecteur. L'Algorithme 2 montre une boucle multidimensionnelle mise à plat (ligne n° 1 de l'Algorithme 1, avec $d = \dim(\mathbf{r}) = \dim(\mathbf{s}_{\text{répétition}})$, $\mathbf{r} = \begin{pmatrix} r_0 \\ \cdots \\ r_{d-1} \end{pmatrix}$ et $\mathbf{s}_{\text{répétition}} = \begin{pmatrix} s_0 \\ \cdots \\ s_{d-1} \end{pmatrix}$.

Algorithme 1 Traduction d'une répétition vers des boucles multidimensionnelles.

1: **for** $\mathbf{r} = \mathbf{0}$ to $\mathbf{s}_{\text{répétition}}$ **do** /* pour toutes les répétitions */
2: **for** $k = 0$ to n **do** /* pour touts les tilers d'entrée */
3: **for** $\mathbf{i} = \mathbf{0}$ to $\mathbf{s}_{\text{motif\_opérande}_k}$ **do** /* pour tous les indices du motif */
4: /* calcule l'indice dans le tableau d'entrée */
5: $\mathbf{e}_{\mathbf{r_i}} = \mathbf{o}_{\mathbf{e_k}} + (P_{e_k}\ F_{e_k}) \cdot \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s}_{\text{tableau\_entrée}_k}$
6: /* copie l'élément du tableau dans le motif opérande */
7: $\text{Motif\_Opérande}_k[\mathbf{i}] = \text{Tableau\_entrée}_k[\mathbf{e}_{\mathbf{r_i}}]$
8: **end for**
9: **end for**
10: /* exécute le calcul de la tâche répétée */
11: $\text{Motif\_Résultat}_{h(0 \leqslant h < m)} = \text{Traitement}(\text{Motif\_Opérande}_{k(0 \leqslant k < n)})$
12: **for** $h = 0$ to m **do** /* pour touts les tilers de sortie */
13: **for** $\mathbf{i} = \mathbf{0}$ to $\mathbf{s}_{\text{motif}_h}$ **do** /* pour tous les indices du motif */
14: /* calcule l'indice dans le tableau de sortie */
15: $\mathbf{e}_{\mathbf{r_i}} = \mathbf{o}_{\mathbf{s_h}} + (P_{s_h}\ F_{s_h}) \cdot \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s}_{\text{tableau\_sortie}_h}$
16: /* copie l'élément du motif résultat dans le tableau */
17: $\text{Tableau\_sortie}_h[\mathbf{e}_{\mathbf{r_i}}] = \text{Motif\_Résultat}_h[\mathbf{i}]$
18: **end for**
19: **end for**
20: **end for**

Algorithme 2 Mis-à-plat d'une boucle multidimensionnelle.

1: **for** $r_0 = 0$ to s_0 **do**
2: **for** $r_1 = 0$ to s_1 **do**
3: ... /* boucles de 2 à $d-2$ */
4: **for** $r_{d-1} = 0$ to s_{d-1} **do**
5: ... /* le corps de la boucle */
6: **end for**
7: **end for**
8: **end for**

DESCRIPTION COMPOSÉE

Cette description se traduit sous la forme de boucles comme une succession de nids de boucles représentant la traduction de chaque appel aux sous-tâches.

Observation. La succession doit correspondre à l'ordre partiel strict des appels.

DESCRIPTION HIÉRARCHIQUE

Une description hiérarchique se traduit par la hiérarchisation des boucles : le traitement des motifs (la ligne n° 11 de l'Algorithme 1 est remplacée par la traduction par des boucles du niveau inférieur de la hiérarchie.

ANALYSE

Nous avons vu comment une description MARTE RSM peut se traduire naturellement sous la forme des nid de boucles. En plus, ces nids de boucles ont des caractéristiques qui les rendent parfaits pour des techniques d'optimisation basées sur des transformations de boucles. Néanmoins, les transformations de boucles sont des optimisations source à source, et une éventuelle stratégie d'optimisation de la spécification MARTE RSM par la traduction sous la forme de nids de boucles et l'utilisation des transformations de boucles n'est pas le bon choix :

- premièrement, les boucles résultat après telles transformations ne peuvent pas être retraduites directement sous la forme d'une spécification MARTE RSM ;
- deuxièmement, des informations structurelles issues du formalisme répétitive sont perdues (parallélisme, assignation unique, etc.).

C'est une raison pour laquelle l'utilisation des transformations de plus haut niveau qui ressemblent aux transformations de boucles, mais s'appliquent au niveau de la spécification multidimensionnelle (sur des répétitions data-parallèles), a été choisie.

Observation. L'utilisation de telles transformations de haut niveau n'interdit pas l'utilisation des transformations de boucles au niveau d'abstraction plus proche à l'exécution, tout comme au niveau de la génération du code ou de la compilation.

Pour appliquer les optimisations, il nous faut donc connaître les dépendances de données présentes dans l'application. En effet, la cohésion des dépendances est nécessaire pour qu'une transformation ne modifie pas les résultats d'une application. Il existe plusieurs modèles pour représenter et manipuler les dépendances de données, mais l'utilisation de tableaux toriques dans Array-OL complexifie grandement la manipulation des dépendances, car elle les rend non linéaires. Nous ne détaillerons donc pas ici les techniques habituelles, toutefois le lecteur trouvera des études très détaillées dans [37, 21].

La manipulation des dépendances est le réel défi de la transformation d'applications Array-OL. Il est pour cette raison que TUS a proposé son propre formalisme de manipulations de dépendances : les ODT.

5.2 FORMALISME ODT

Le formalisme ODT (Opérateurs de Description de Tableau) a été proposé par Demeure [27] afin d'exprimer les dépendances entre les parties opérandes et résultats d'une tâche Array-OL. Il est basé sur l'algèbre linéaire avec contraintes et est constitué de plusieurs opérateurs définissant les liens entre deux espaces $\mathbb{K}^n$. On peut relier ces opérateurs donc ces espaces à l'aide d'une loi de composition. Pour transcrire en ODT les dépendances d'une tâche Array-OL, il faut identifier les points des tableaux à leurs coordonnées. Puis c'est sur ces coordonnées que sont appliqués les opérateurs. On obtient ainsi une suite d'opérateurs exprimant les dépendances. Il est alors possible d'effectuer un certain nombre de calculs découlant des propriétés intrinsèques à la description répétitive.

Le formalisme ODT et ses opérateurs sont étudiés en détail par Soula en [85] et Dumont en [32]. Par la suite nous allons faire un court résumé du formalisme et, plus important, de la représentation d'une répétition sous cette forme.

5.2.1 *Opérateurs* ODT

Chaque opérateur est en fait une relation binaire de $\mathbb{K}^n$ dans $\mathbb{K}^m$ et se comprend par une lecture de la droite vers la gauche. Dans le contexte particulier d'ARRAY-OL, nous utiliserons le plus souvent $\mathbb{Z}$ comme espace, car les opérateurs sont appliqués aux coordonnées de points d'un tableau.

Au total, il existe neuf opérateurs :

MODULO : Une valeur infinie dans le modulo indique que seules les valeurs positives sont gardées. Le modulo se note $\left(\overrightarrow{m}\right)_{\mathbf{M}}$.

$$\forall \overrightarrow{m} \in (\mathbb{Z}^*)^n, \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^n, \quad \overrightarrow{y} M \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \overrightarrow{x} \mod \overrightarrow{m}$$

SHIFT : Il s'agit simplement d'une opération de translation. Elle se note $\left(\overrightarrow{shift}\right)_{\mathbf{S}}$.

$$\forall \overrightarrow{shift} \in \mathbb{K}^n, \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^n, \quad \overrightarrow{y} S \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \overrightarrow{x} + \overrightarrow{shift}$$

GABARIT : Le gabarit agit comme un filtre laissant passer certains points et en bloquant d'autres. Il se note $\left(\overrightarrow{min}, \overrightarrow{max}\right)_{\mathbf{G}}$ ou simplement $\left(\overrightarrow{max}\right)_{\mathbf{G}}$ si $\overrightarrow{min}$ est nul. En outre une valeur infinie indique une absence de contrainte.

$$\forall (\overrightarrow{min}, \overrightarrow{max}) \in \mathbb{Z}^n \times \mathbb{Z}^n, \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^n, \quad \overrightarrow{y} G \overrightarrow{x} \Leftrightarrow \overrightarrow{min} \leqslant \overrightarrow{x} < \overrightarrow{max} \text{ et } \overrightarrow{x} = \overrightarrow{y}$$

PROJECTION : La projection est une multiplication matricielle et permet donc d'effectuer des changements de repères ou de calculer le résultat d'applications linéaires. Elle se note $\left|\mathcal{M}\right|$. La taille de l'espace de départ est égale aux nombre de colonnes de $\mathcal{M}$ alors que le nombre de lignes est égale à la taille de l'espace d'arrivée.

$$\forall \mathcal{M} \in M_{mn}(K), \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^m, \quad \overrightarrow{y} \text{ Pro } \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \mathcal{M}.\overrightarrow{x}$$

SEGMENTATION : La matrice $\mathcal{M}$ d'une projection n'étant pas forcément inversible la segmentation sert de notation pour exprimer une « inversion théorique ». Elle se note $\overline{\mathcal{M}}$. La concordance entre le nombre de lignes et de colonnes avec la dimension des espaces est inversée par rapport à la projection.

$$\forall \mathcal{M} \in M_{mn}(K), \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^m, \quad \overrightarrow{y} \text{ Seg } \overrightarrow{x} \Leftrightarrow \overrightarrow{x} = \mathcal{M}.\overrightarrow{y}$$

ÉCLATEMENT : Une valeur infinie dans l'éclatement indique que seules les valeurs positives sont gardées. L'éclatement se note $\left(\overrightarrow{m}\right)_{\star}$.

$$\forall \overrightarrow{m} \in (\mathbb{Z}^*)^n, \forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{K}^n \times \mathbb{K}^n, \overrightarrow{0} \leqslant \overrightarrow{x} < \overrightarrow{m}, \quad \overrightarrow{y} E \overrightarrow{x} \Leftrightarrow \forall \overrightarrow{k} \in \mathbb{K}^n, \overrightarrow{y} = \overrightarrow{x} + \overrightarrow{k} \times \overrightarrow{m}$$

ARRONDI : L'arrondi se note $\lfloor . \rfloor$.

$$\forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{Q}^n \times \mathbb{Z}^n, \quad \overrightarrow{y} A \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \lfloor \overrightarrow{x} \rfloor$$

FRACTIONNEUR : Le fractionneur se note $\lfloor . \rfloor$.

$$\forall (\overrightarrow{x}, \overrightarrow{y}) \in \mathbb{Z}^n \times \mathbb{Q}^n, \exists \overrightarrow{r} \in \mathbb{Q}^n, \overrightarrow{0} \leqslant \overrightarrow{r} < \overrightarrow{1}, \quad \overrightarrow{y} F \overrightarrow{x} \Leftrightarrow \overrightarrow{y} = \overrightarrow{x} + \overrightarrow{r}$$

5.2.2 *Propriétés des* ODT

Il existe une *loi de composition* sur les ODT et elle est identique à celle des relations. Elle ne permet donc de composer que des ODT qui ont des espaces d'arrivée et de départ communs. Elle se lit de droite à gauche et se note « . ».

Par symétrie, nous désignons le fait de construire un ODT ayant exactement les mêmes liens entre les points des deux espaces extrêmes, mais en échangeant les espaces source et destination. Nous appellerons indifféremment *miroir* ou *symétrique*, l'ODT résultat de la symétrie et le désignerons à l'aide de l'exposant $^{-1}$.

Le miroir d'une composition d'ODT s'effectue de la même manière que l'inverse d'une composition de fonction : en composant les miroirs des ODT dans l'ordre opposé

$$(\mathrm{ODT}_1.\mathrm{ODT}_2)^{-1} = \mathrm{ODT}_2^{-1}.\mathrm{ODT}_1^{-1} \tag{5.2}$$

Il nous suffit donc de décrire les miroirs des opérateurs élémentaires :

GABARIT $\left(\overrightarrow{min}, \overrightarrow{max}\right)_{\mathbf{G}}{}^{-1} = \left(\overrightarrow{min}, \overrightarrow{max}\right)_{\mathbf{G}}$

DÉCALAGE $\left(\overrightarrow{shift}\right)_{\mathbf{S}}{}^{-1} = \left(-\overrightarrow{shift}\right)_{\mathbf{S}}$

MODULO $\left(\overrightarrow{m}\right)_{\mathbf{M}}{}^{-1} = \left(\overrightarrow{m}\right)_{\star}$

ÉCLATEMENT $\left(\overrightarrow{m}\right)_{\star}{}^{-1} = \left(\overrightarrow{m}\right)_{\mathbf{M}}$

PROJECTION $\left|\mathcal{M}\right|^{-1} = \overline{\underline{\mathcal{M}}}$

SEGMENTATION $\overline{\underline{\mathcal{M}}}{}^{-1} = \left|\mathcal{M}\right|$

ARRONDI $\lfloor^{-1} = \rfloor$

FRACTIONNEUR $\rfloor^{-1} = \lfloor$

5.2.3 *Représentation d'une répétition en* ODT

Le formalisme ODT est conçu pour exprimer les dépendances de données entre les éléments des tableaux résultats et ceux opérandes d'une description Array-OL. Dans une décomposition répétitive, les dépendances sont identifiables au niveau des motifs d'entrée/sortie :

A. l'ensemble d'éléments d'un motif de sortie dépend de tous les éléments des motifs d'entrée de la même instance de la répétition ;

B. le lien entre les éléments des tableaux d'entrée et de sortie d'une répétition se fait en passant par l'espace de répétition.

Dans le cas de la décomposition composée ou hiérarchique, le lien se fait par les éléments des tableaux connectés par des liens.

Nous sommes surtout intéressées de la modélisation des dépendances d'une répétition et cela est possible en utilisant les opérateurs ODT entre les espaces multidimensionnels qui définissent les formes des tableaux d'entrée, de sortie et l'espace de répétition.

Représentation des accès par motifs uniformes

Pour modéliser en ODT les correspondances entre les éléments d'un tableau et l'ensemble des motifs qui correspondent à l'espace de répétitions, nous allons passer par la description du tiler associé, pour faire le lien entre l'espace de points du tableau, M, caractérisé par sa forme multidimensionnelle et l'ensemble espace d'itération Q et l'espace D du motif, ce qu'on note comme espace QD. L'Équation 5.3 contient la relation ODT qui exprime cette correspondance et est dérivée directement de la définition des tilers (*cf* l'Équation 5.1).

$$\left(\mathrm{M}\right)_{\mathbf{M}} . \left(\mathrm{O}\right)_{\mathbf{S}} . \left|\mathrm{P} \quad \mathrm{F}\right| . \begin{pmatrix} \mathrm{Q} \\ \mathrm{D} \end{pmatrix}_{\mathbf{G}} \tag{5.3}$$

Représentation dès dépendances entrées-sorties

Les dépendances entre un tableau opérande et un tableau résultat dans une répétition se fait en passant par l'espace de répétition commun,

$$\begin{aligned} &\left(\mathrm{M}_{op}\right)_{\mathbf{M}} . \left(\mathrm{O}_{op}\right)_{\mathbf{S}} . \left|\mathrm{P}_{op} \quad \mathrm{F}_{op}\right| . \begin{pmatrix} \mathrm{Q} \\ \mathrm{D}_{op} \end{pmatrix}_{\mathbf{G}} \\ &\left(\mathrm{M}_{res}\right)_{\mathbf{M}} . \left(\mathrm{O}_{res}\right)_{\mathbf{S}} . \left|\mathrm{P}_{res} \quad \mathrm{F}_{res}\right| . \begin{pmatrix} \mathrm{Q} \\ \mathrm{D}_{res} \end{pmatrix}_{\mathbf{G}} , \end{aligned} \tag{5.4}$$

en inversant la relation résultat (*cf* l'Équation 5.2 et les règles d'inversion) et les ajustant vers un espace commun $\begin{pmatrix} \mathrm{Q} \\ \mathrm{D}_{op} \\ \mathrm{D}_{res} \end{pmatrix}$:

$$\begin{aligned} &\left(\mathrm{M}_{op}\right)_{\mathbf{M}} . \left(\mathrm{O}_{op}\right)_{\mathbf{S}} . \left|\mathrm{P}_{op} \quad \mathrm{F}_{op} \quad 0\right| . \begin{pmatrix} \mathrm{Q} \\ \mathrm{D}_{op} \\ \mathrm{D}_{res} \end{pmatrix}_{\mathbf{G}} \\ &\begin{pmatrix} \mathrm{Q} \\ \mathrm{D}_{op} \\ \mathrm{D}_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\underline{\mathrm{P}_{res} \quad 0 \quad \mathrm{F}_{res}}} . \left(-\mathrm{O}_{res}\right)_{\mathbf{S}} . \left(\mathrm{M}_{res}\right)_{\star} . \end{aligned} \tag{5.5}$$

Par la composition des deux relations, la relation qui exprime les dépendances entre les éléments d'un tableau opérande et un tableau résultat devient :

$$\left(M_{op}\right)_{\mathbf{M}} . \left(O_{op}\right)_{\mathbf{S}} . \left|\begin{matrix} P_{op} & F_{op} & 0 \end{matrix}\right| . \begin{pmatrix} Q \\ D_{op} \\ D_{res} \end{pmatrix}_{\mathbf{G}} . \overline{\begin{matrix} P_{res} & 0 & F_{res} \end{matrix}} . \left(O_{res}\right)_{\mathbf{S}} . \left(M_{res}\right)_{\star} \tag{5.6}$$

Représentation d'une répétition complète

Une répétition complète peut consommer plusieurs tableaux opérandes et en produire plusieurs. En gardant la même logique que pour l'expression des dépendances entre un tableau d'entrée et un de sortie, la relation qui décrit une répétition complète devient :

$$\begin{pmatrix} M_{op_1} \\ M_{op_2} \\ \vdots \end{pmatrix}_{\mathbf{M}} . \begin{pmatrix} O_{op_1} \\ O_{op_2} \\ \vdots \end{pmatrix}_{\mathbf{S}} . \left|\begin{matrix} P_{op_1} & F_{op_1} & 0 & 0 & 0 & 0 & 0 \\ P_{op_2} & 0 & F_{op_2} & 0 & 0 & 0 & 0 \\ \vdots & 0 & 0 & \ddots & 0 & 0 & 0 \end{matrix}\right| . \begin{pmatrix} Q \\ D_{op_1} \\ D_{op_2} \\ \vdots \\ D_{res_1} \\ D_{res_2} \\ \vdots \end{pmatrix}_{\mathbf{G}}$$
$$\overline{\begin{matrix} P_{res_1} & 0 & 0 & 0 & F_{res_1} & 0 & 0 \\ . P_{res_2} & 0 & 0 & 0 & 0 & F_{res_2} & 0 \\ \vdots & 0 & 0 & 0 & 0 & 0 & \ddots \end{matrix}} . \begin{pmatrix} O_{res_1} \\ O_{res_2} \\ \vdots \end{pmatrix}_{\mathbf{S}} . \begin{pmatrix} M_{res_1} \\ M_{res_2} \\ \vdots \end{pmatrix}_{\star} \tag{5.7}$$

Le formalisme ODT *n'est qu'un outil d'expression des dépendances permettant d'effectuer un certain nombre d'opérations qui se traduisent sous la forme des transformations* ODT.

Observation. Il n'y a pas une équivalence totale entre la représentation répétitive et ODT. Dans l'Équation 5.7, les tableaux ne sont plus différentiables par la présence des zéros et ainsi cette expression ne permet pas de revenir à sa description répétitive sans des informations supplémentaires.

« Court-circuit »

Une construction intéressante est celle qu'on appelle le « court-circuit », qui permet dans une hiérarchie d'exprimer les coordonnées des éléments d'un motif d'une sous-tâche selon les coordonnées des éléments du tableau de la tâche supérieure. On note de la façon suivante les représentations ODT des tâches supérieure et inférieure :

$$\left(M_{up}\right)_{\mathbf{M}} . \left(O_{up}\right)_{\mathbf{S}} . \left|\begin{matrix} P_{up} & F_{up} \end{matrix}\right| . \begin{pmatrix} Q_{up} \\ D_{up} \end{pmatrix}_{\mathbf{G}} \tag{5.8}$$

$$\left(M_{sub}\right)_{\mathbf{M}} . \left(O_{sub}\right)_{\mathbf{S}} . \left|\begin{matrix} P_{sub} & F_{sub} \end{matrix}\right| . \begin{pmatrix} Q_{sub} \\ D_{sub} \end{pmatrix}_{\mathbf{G}} \tag{5.9}$$

On peut alors écrire que les coordonnées, pour des itérations de pavage et d'ajustage fixés, sont de la forme :

$$\left(O_{up}\right)_{\mathbf{S}} . \left|\begin{matrix} P_{up} & F_{up} \end{matrix}\right| . \begin{pmatrix} X_{q,up} \\ O_{sub} + \left|\begin{matrix} P_{sub} & F_{sub} \end{matrix}\right| . \begin{pmatrix} X_q \\ X_d \end{pmatrix}_{\mathbf{G}} \end{pmatrix}_{\mathbf{G}} \tag{5.10}$$

On obtient donc des coordonnées égales à :

$$O_{up} + P_{up}X_{q,up} + F_{up}O_{sub} + \left|F_{up}P_{sub} \quad F_{up}F_{sub}\right| \begin{pmatrix} X_q \\ X_d \end{pmatrix}_{\mathbf{G}} \tag{5.11}$$

Il est alors possible de modifier la forme de nos deux tâches, en passant directement à la sous-tâche, un tableau qui corresponde aux motifs qu'elle consomme.

$$\left(O_{up} + F_{up}O_{sub}\right)_{\mathbf{S}} \cdot \left|P_{up} \quad F_{up}P_{sub} \quad F_{up}F_{sub}\right| \cdot \begin{pmatrix} Q_{up} \\ Q_{sub} \\ D_{sub} \end{pmatrix}_{\mathbf{G}} \tag{5.12}$$

Par l'enchaînement des constructions de court-circuit on peut exprimer les coordonnées des éléments d'un tableau à un niveau quelconque en fonction des éléments d'un tableau à un niveau supérieur.

5.2.4 *Utilisation des* ODT

Le formalisme ODT à été conçu pour exprimer les relations entre les espaces multidimensionnels dans une décomposition répétitive qui est au cœur de la représentation Array-OL et MARTE RSM. En plus, en appliquant des opérations définies sur le formalisme, la représentation en ODT des répétitions peuvent être transformées sous une forme équivalente (les mêmes dépendances exactes sont exprimées) mais qui correspond à une représentation répétitive différente.

Une succession d'opérations ODT qui, appliquée sur la représentation en ODT d'une structure répétitive, arrive à la convertir dans une autre représentation qui correspond (et peut être traduite) à une représentation répétitive correcte représente une transformation ODT. Une telle transformation, par la propagation des opérateurs ODT, garanti la conformité de la transformation.

Une transformation de haut niveau a associée une transformation ODT qui formalise son fonctionnement, garanti sa conformité et amène des informations de gain, telles que la minimisation des tableau intermédiaires.

L'exécution d'une transformation Array-OL peut être décomposée normalement dans les étapes suivantes :

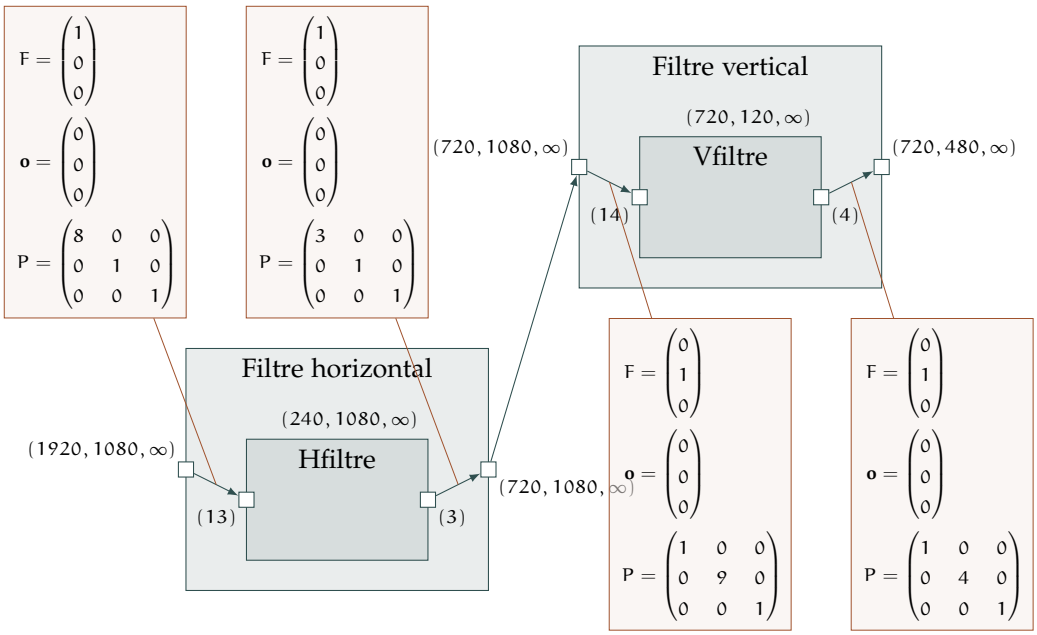
1. identification des éléments structurels qui participent dans la transformation : normalement des répétitions successives ou hiérarchiques ;
2. transformation de ces éléments sous la forme ODT ;
3. application des successions d'opérateurs ODT qui représentent la transformation ODT ;
4. convertir la représentation ODT résultat sous la forme répétitive ;
5. réinsertion des éléments transformés dans la spécification complète.

Nous n'allons pas montrer le fonctionnement complet des transformations ODT. Elles sont décrites en détail et prouvées dans les thèses de Soula [85] et Dumont [32]. Par la suite nous allons faire une liste des transformations Array-OL disponibles, montrant très sommaire la correspondance en termes d'ODT.

5.3 TRANSFORMATIONS DATA-PARALLÈLES

Les transformations de haut niveau data-parallèles permettent la restructuration d'une spécification répétitive, par la redistribution des répétitions entre les niveaux de la hiérarchie. Dans cette section nous allons étudier les différentes transformations qui sont disponibles. Ces transformations ont un fonctionnement similaire aux transformations de boucles homonymes, sur lesquelles elles sont basées.

Nous allons illustrer le fonctionnement de chaque transformation sur l'application de *Downscaler* que nous avons utilisée pour montrer la modélisation répétitive d'Array-OL dans la section 2.1. La décomposition complète de l'application en deux filtres successifs est montrée sur la Figure 54.



L'application est décomposée en deux filtres successifs, un qui réduit chaque image sur la dimension horizontale, *Filtre horizontal*, et l'autre qui réduit ensuite l'images sur la dimension verticale, *Filtre vertical*. Chaque filtre a une fonctionnalité répétitive, décrite par l'espace de répétition de chaque répétition et avec des accès uniformes définis par les tilers :

- la tâche répétée du filtre horizontal prend une fenêtre de 13 éléments successifs qui glisse sur chaque ligne avec un pas de 8 et produit 3 éléments qui sont rangés dans le tableau de sortie un après l'autre, pour toutes les lignes et toutes les images ;
- la tâche répétée du filtre vertical a un fonctionnement similaire, mais cette fois une fenêtre de glissement de 14 avec un pas de 9 qui produit 4 éléments, pour chaque colonne de chaque image.

FIG. 54: Filtres répétitifs de l'application de Downscaler

5.3.1 *Fusion*

Définition 5 (Fusion élémentaire)**.** La transformation de fusion prend deux répétitions *successives* et calcule une répétition commune pour les deux, pendant que les sous-répétitions qui restent sont placées au niveau inférieur de la hiérarchie, en minimisant le tableau intermédiaire entre ces deux sous-répétitions. Un des effets de la fusion est l'introduction d'un niveau de répétition dans la hiérarchie.

Définition 6 (Tâches successives)**.** Deux tâches sont considérées successives si elles se trouvent au même niveau hiérarchique et la première produit un tableau consommé par la seconde. La propriété de succession est en lien direct avec l'ordre partiel strict des tâches.

Définition 7 (Tableau intermédiaire minimal)**.** Entre deux répétitions successives, un tableau intermédiaire minimal est désigné par un groupe minimal de motifs produit par la première tâche qui permet à la deuxième répétition de s'exécuter au moins une fois, donc produire des éléments et en conséquence permettant une exécution sans blocage.

Exemple. La Figure 55 montre le résultat de la fusion sur les deux répétitions successives de la modélisation du Downscaler de la Figure 54. Nous pouvons observer la création de la répétition commune et le déplacement des deux tâches répétées à un niveau inférieur de la hiérarchie, avec des espaces de répétition réduits et en consommant des tableaux avec des dimensions réduites.

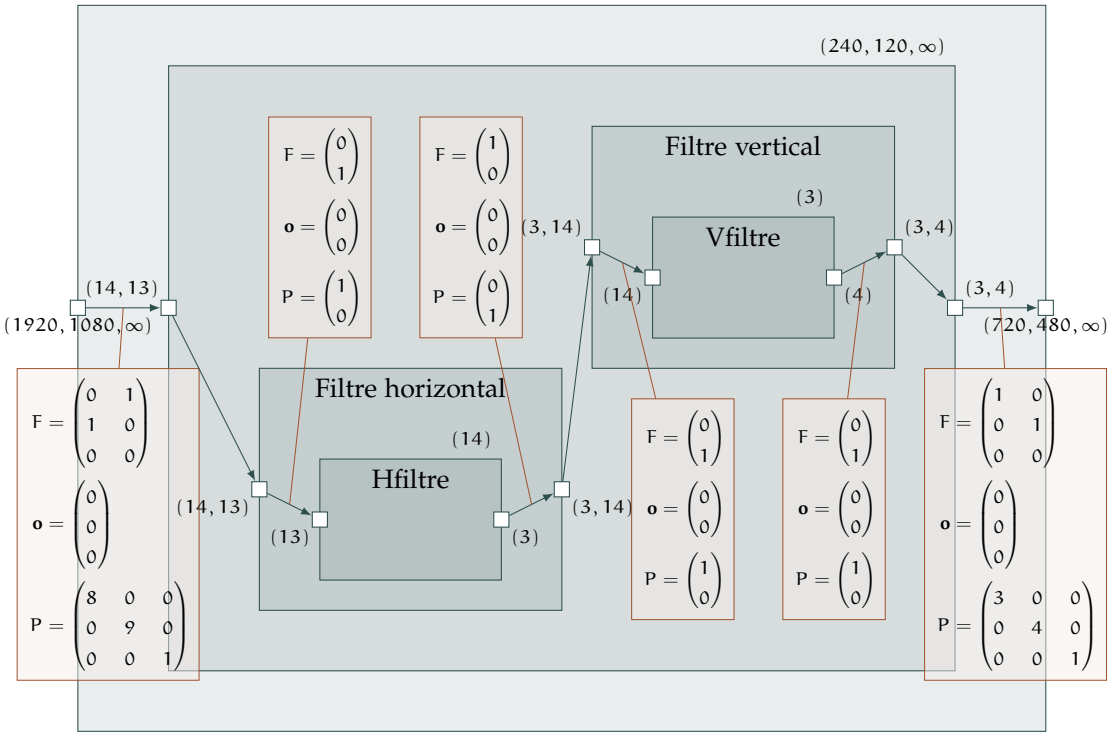


FIG. 55: Downscaler après la fusion

CALCUL DE LA FUSION

Le but de la fusion est la minimisation du tableau intermédiaire et donc de trouver des macromotifs produits par la première répétition qui permettent à la deuxième tâche de s'exécuter : le macromotif produit par la première tâche contient des motifs entiers consommés par la seconde.

Définition 8 (Macromotif)**.** L'agglomération de plusieurs motifs, toujours sous une forme uniforme.

Dans ce but, le formalisme d'ODT et sa capacité à exprimer les dépendances sont utilisées. Nous n'allons pas montrer toutes les étapes

de calcul de la fusion en ODT, qui sont présentées en détail dans la thèse de Dumont. Nous allons nous contenter de montrer le principe et les résultats.

En commençant avec les représentations ODT des deux répétitions successives,

$$T_1 \mapsto \left(M_1\right)_{\mathbf{M}} \cdot \left(S_1\right)_{\mathbf{S}} \cdot \left|P_{op_1} \quad F_{op_1} \quad 0\right| \cdot \begin{pmatrix} Q_1 \\ D_{op_1} \\ D_{res_1} \end{pmatrix}_{\mathbf{G}} \cdot \overline{\underline{P_{res_1} \quad 0 \quad F_{res_1}}} \tag{5.13}$$

$$T_2 \mapsto \left(M_2\right)_{\mathbf{M}} \cdot \left(S_2\right)_{\mathbf{S}} \cdot \left|P_{op_2} \quad F_{op_2} \quad 0\right| \cdot \begin{pmatrix} Q_2 \\ D_{op_2} \\ D_{res_2} \end{pmatrix}_{\mathbf{G}} \cdot \overline{\underline{P_{res_2} \quad 0 \quad F_{res_2}}} \tag{5.14}$$

et suivant la loi de composition entre les deux relations (opération permise par la présence du du tableau commun, tableau résultat de la première et opérande de la seconde), nous nous retrouvons avec la relation

$$\left(M_1\right)_{\mathbf{M}} \cdot \left(S_1\right)_{\mathbf{S}} \cdot \left|P_{op_1} \quad F_{op_1} \quad 0\right| \cdot \underbrace{\begin{pmatrix} Q_1 \\ D_{op_1} \\ D_{res_1} \end{pmatrix}_{\mathbf{G}} \cdot \overline{\underline{P_{res_1} \quad 0 \quad F_{res_1}}} \cdot \left(M_2\right)_{\mathbf{M}} \cdot \left(S_2\right)_{\mathbf{S}} \cdot \left|P_{op_2} \quad F_{op_2} \quad 0\right| \cdot \begin{pmatrix} Q_2 \\ D_{op_2} \\ D_{res_2} \end{pmatrix}_{\mathbf{G}}}_{\text{relation } Q_1 D_1 \to Q_2 D_2 \text{ passant par } A_2}$$

$$\cdot \overline{\underline{P_{res_2} \quad 0 \quad F_{res_2}}} \cdot \tag{5.15}$$

À partir la partie centrale de l'Équation 5.15 qui exprime le passage entre l'espace $\begin{pmatrix} Q_1 \\ D_1 \end{pmatrix}$ et $\begin{pmatrix} Q_2 \\ D_2 \end{pmatrix}$ en passant par le tableau intermédiaire M_2, une succession d'opérateurs ODT est appliquée pour calculer l'espace de répétition commun et les macromotifs opérandes et résultats, ce que se traduit par trois relations équivalentes à des expressions en ODT des trois répétitions :

– pour la tâche supérieure :

$$\left(M_1\right)_{\mathbf{M}} \cdot \left(S_{up}\right)_{\mathbf{S}} \cdot \left|P_{op_{up}} \quad F_{op_{up}} \quad F_{op_1} \quad 0 \quad 0\right| \cdot \begin{pmatrix} Q \\ D_{\mathcal{M}_{op}} \\ D_{op_1} \\ D_{\mathcal{M}_{res}} \\ D_{res_2} \end{pmatrix}_{\mathbf{G}} \cdot \overline{\underline{P_{res_{up}} \quad 0 \quad 0 \quad F_{res_{up}} \quad F_{res_2}}} \ ;$$

– pour la première sous-tâche :

$$\begin{pmatrix} D_{\mathcal{M}_{op}} \\ D_{op_1} \end{pmatrix}_{\mathbf{M}} \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix}_{\mathbf{S}} \cdot \left|\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{matrix}\right| \cdot \begin{pmatrix} D_{\mathcal{M}_{op}} \\ D_{op_1} \\ D_{res_1} \end{pmatrix}_{\mathbf{G}} \cdot \overline{\underline{P_{res_{sub_1}} \quad 0 \quad F_{res_1}}} \cdot \left(S_{res_{sub_1}}\right)_{\mathbf{S}} \cdot \left(M_2\right)_{\star} \ ;$$

– pour la deuxième sous-tâche :

$$\left(M_2\right)_{\mathbf{M}} \cdot \left(S_{op_{sub_2}}\right)_{\mathbf{S}} \cdot \left|P_{res_{sub_2}} \quad F_{op_2} \quad 0\right| \cdot \begin{pmatrix} D_{\mathcal{M}_{res}} \\ D_{op_2} \\ D_{res_2} \end{pmatrix}_{\mathbf{G}} \cdot \overline{\underline{\begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}}} \ .$$

$D_{\mathcal{M}_{op}}$ représente le groupe minimal de motifs initiaux qui forment les macromotifs du M_2. Le même macromotif est consommé par la deuxième sous-répétition sous la forme d'un groupe de motifs $D_{\mathcal{M}_{res}}$.

PLUSIEURS TABLEAUX INTERMÉDIAIRES

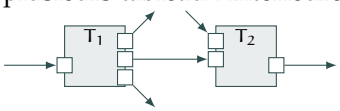
La transformation de fusion élémentaire présentée est formalisée pour le cas de répétitions avec un seul tableau d'entrée et de sortie et un tableau intermédiaire unique entre les deux répétitions :

.

Dumont propose en [32] des solutions pour d'autres cas possible. Il distingue cinq cas différents :

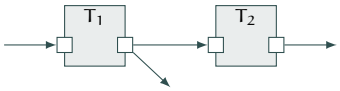
1. plusieurs tableaux extérieures ;

2. plusieurs tableaux intérieures ;

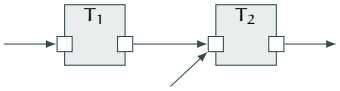


3. plusieurs tableaux intermédiaires ;

4. double consomption du tableau intermédiaire ;



5. double production du tableau intermédiaire.



Le 5^{e} cas est interdit par la propriété d'assignation unique d'ARRAY-OL et à part le 3^{e} cas, avec plusieurs tableaux intermédiaires, les autres sont traités dans une manière similaire. On peut observer que dans tous ces cas il y a un seul tableau intermédiaire et indifféremment de nombre de tableaux extérieurs ou intérieurs des deux tâches, le calcul du macromotif dépend exclusivement des motifs de production/consomption du ce tableau. Dans sa thèse, Soula préfère la description complète des tâches (*cf* l'Équation 5.7) mais Dumont argumente que ce choix complexifie beaucoup l'implémentation, en rendent plus complexe la distinction des différentes tâches dans la forme ODT. Dumont opte pour l'application du calcul du macromotif individuellement pour chacun des tableaux, sachant que le macromotif restera le même. Cela facilite surtout la séparation et l'identification des parties ODT qui correspondent à chacun des tableaux après le calcul de la fusion.

Quant au 3^{e} cas, la présence de plusieurs tableaux intermédiaires, une solution idéale pour la fusion n'existe pas encore. La représentation ODT complète de plusieurs tableaux intermédiaires n'est pas possible puisqu'elle rendrait non exacte la partie droite de la première répétition, ce qui bloquerait l'inversion du ODT exact.

Dumont propose d'effectuer la fusion en plusieurs étapes, en différenciant la production des tableaux intermédiaires par plusieurs copies de

la première tâche et effectuant plusieurs fusions, pour chaque tableau intermédiaire à la fois. Un tel choix aura comme résultat la génération d'un nombre de niveaux de hiérarchie égal au nombre de tableaux intermédiaires. À part l'explosion en niveaux hiérarchiques, cela impliquerait aussi l'exécution multiple de la première répétition, ce qu'on appelle apparition des recalculs.

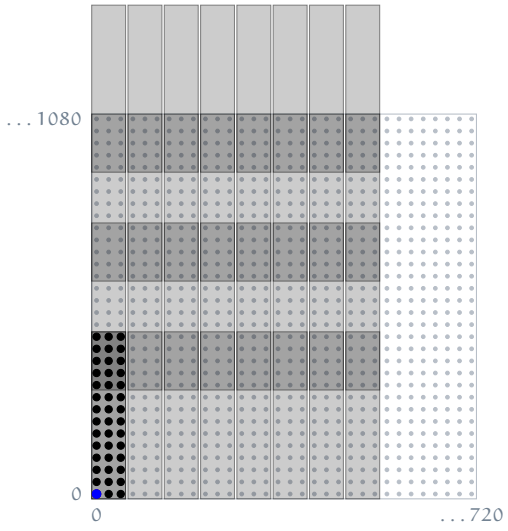
RECALCULS

Définition 9 (Recalcul)**.** Les recalculs sont représentés par l'augmentation de la répétition complète pour la première tâche impliquée dans une fusion, causée par les motifs de production/consomption entre les deux répétitions.

Observation. La répétition complète d'une tâche (*cf* la section 2.4, Définition statique) est donnée par la concaténation de toutes les formes multidimensionnelles en descendant les niveaux de la hiérarchie du haut jusqu'au niveau de la tâche concernée. Recalculs sont représentés par l'augmentation de la répétition complète pour une tâche, suite à une transformation de fusion.

Le recalcul intervient lorsque deux macromotifs partagent des motifs originaux (ou des parties de motifs) en commun, ce que fait que deux exécutions de la première sous-répétition produit plusieurs fois les mêmes éléments et donc augmente la quantité de calculs de la spécification, même si la fonctionnalité décrite reste la même.

Exemple. Le cas le plus souvent de recalculs rencontré en pratique est causé par des accès en fenêtre glissante, dans la deuxième répétition. Il est le cas de l'application de Downscaler où la deuxième répétition a un tel motif d'accès, comment on peut voir sur la Figure 56.



Le placement des macromotifs dans le tableau intermédiaire initial montre le chevauchement entre les macromotifs successifs sur la dimension du glissement. Tous les éléments qui sont partagés entre des macromotifs (en gris foncé sur la figure) seront calculés deux fois par la première sous-répétition après la fusion.

FIG. 56: Chevauchement du macromotif dans le tableau initial

Avant la fusion, *cf* la Figure 54, nous avons pour la première répétition un espace de répétition avec une forme de $(240, 1080, \infty)$, ce que représente $240 \times 1080 = 259200$ exécutions de la tâche répétée pour chaque image. Après la fusion, *cf* la Figure 55, l'espace de répétition total pour la première sous-répétition, en concaténant les répétitions

des deux niveaux de la hiérarchie, devient $(240, 120, \infty, 14)$, et donc $240 \times 120 \times 14 = 403200$ exécutions pour chaque image. Le facteur de recalcul introduit par la fusion[1] est de $\approxeq 1.56$ pour la première répétition. Ce facteur s'explique par le partage de 6 lignes entre chaque deux macromotifs successifs contenant 14 lignes, qui seront calculées deux fois par la première sous-répétition.

[1] En divisant les deux quantités d'exécutions.

RÉDUCTION DES TABLEAUX INTERMÉDIAIRES

La réduction des tableaux intermédiaires par le calcul d'un macromotif minimal est une opération de base dans le cadre du refactoring ARRAY-OL.

Des techniques d'optimisation de stockage intra-tableau [25, 89, 81] dans le contexte des transformations de boucles permettent le calcul de la « fenêtre de référence » d'un tableau multidimensionnel, qui permet la réutilisation des emplacements mémoires pour les éléments du même tableau. Par une analyse des domaines de définition et opérandes et de l'ordre d'exécution et de stockage [25], les tailles mémoires des tableaux multidimensionnels peuvent être réduites aux ces fenêtres de références.

D'une manière similaire, la fusion ARRAY-OL calcule du macromotif minimal entre deux répétitions qui permet une exécution sans blocage. Le macromotif intermédiaire et les macromotifs d'entrées et de sortie pour les deux sous-répétitions sont les tailles minimales qui doivent être stockées en mémoires si nous considérons une exécution séquentielle pour le niveau de la répétition commune. Si une exécution parallèle est choisie, chaque unité d'exécution doit avoir ses propres macromotifs stockés en mémoire.

Des optimisations pour réduire encore plus la taille globale de la mémoire (ou des mémoires si le système contient plusieurs mémoires distribuées) peuvent être employées avant la compilation par la technique d'optimisation de l'ordre de stockage inter-tableaux : plusieurs tableaux qui ont des durées de vie sans chevauchement partagent des adresses mémoires.

La fusion de deux répétitions successives réduit le tableau intermédiaire entre les deux répétitions. Le problème se complique quand plusieurs répétitions se succèdent et l'objectif devient la réduction de tous les tableaux intermédiaires. La fusion de multiples répétitions sera abordée dans la sous-section 5.4.2.

5.3.2 *Changement du pavage*

Une transformation de changement de pavage agit sur la redistribution de répétitions entre des niveaux successifs de la hiérarchie, de haut en bas. Des répétitions d'un niveau haut de hiérarchie sont descendues au niveau suivant de la hiérarchie, en les ajoutant à chaque répétition de ce niveau.

Tout d'abord, une telle transformation peut être utilisée pour changer la *granularité* de la spécification répétitive. Deuxièmement, elle permet la réduction des recalculs introduits dans une spécification par la fusion.

Définition 10 (Granularité)**.** La notion de *degré de granularité* a été introduite par Labbani *et al.* en [56] dans le contexte du contrôle, qui permet de délimiter les différents cycles d'exécution ou les instants dans lesquels la prise en compte des événements de contrôle devient

possible. Il permet également d'introduire une sémantique de flot dans la description des applications Array-OL pour faciliter l'étude de leur comportement réactif en synchronisant les valeurs des données en entrée avec celles du contrôle. Dans notre contexte, la granularité représente plutôt des sous-ensembles des espaces de répétitions traités comme des blocs à l'exécution.

Exemple. La résultat de la fusion de la Figure 55 détermine le partage de l'application en blocs indépendants, chacun consommant un tableau de $(14, 13)$ éléments et produisant $(3, 4)$. Le degré de granularité de la répétition globale est donné par ces blocs et à l'exécution ils auront associé un ordre d'exécution dicté par les contraintes d'exécution.

Observation. Plusieurs niveaux de répétitions hiérarchiques spécifient plusieurs niveaux de granularité.

Deux transformations de changement de pavage sont disponibles :

PAR AJOUT DE DIMENSIONS. Cette transformation permet de grouper plusieurs motifs du niveau supérieur par le passage de blocs de l'espace de répétition dans le niveau inférieur de la hiérarchie. L'inconvénient est que cela ne réduit pas les recalculs.

PAR AGRANDISSEMENT LINÉAIRE. La transformation est conçue spécialement pour réduire les recalculs introduits dans une application par le calcul des boîtes englobantes qui couvrent plusieurs motifs qui se chevauchent. Elle peut être utilisée uniquement dans le cas des motifs d'accès avec des chevauchements.

Changement de pavage par ajout de dimensions

L'idée pour cette transformation est de descendre des répétitions d'un niveau haut de répétition au niveau qui suit dans la hiérarchie, à toutes les répétitions de ce niveau. L'opération est assez simple, pour passer plusieurs macromotifs au niveau inférieur de la hiérarchie il suffit d'ajouter une dimension dans la forme du motif et de placer plusieurs motifs un après l'autre dans cette dimension.

Définition 11 (Ajout de dimension)**.** Une transformation de changement de pavage par ajout de dimension élémentaire s'applique sur une seule dimension de l'espace de répétition, avec un *facteur* de changement de pavage défini par une valeur naturelle n. Soit $\mathbf{Q} = \left(Q_{i,0\leqslant i<m}\right)$ l'espace de répétition du niveau haut avec m dimensions et k, $0 \leqslant k < m$, l'indice du cet espace sur lequel l'opération de changement de pavage aura lieu. Pour que la transformation soit correcte, n doit diviser exactement la taille de la dimension sur laquelle le changement de pavage s'effectue, $Q_k \mod n = 0$.

Définition 12 (Changement de pavage maximal)**.** Dans le cas où le facteur de changement de pavage coïncide avec la taille de la dimension de la répétition, $Q_k = n$, toute cette dimension de la répétition est passée au niveau inférieur.

Les effets d'une telle transformation, sur les deux niveaux de répétitions hiérarchiques impliquées dans la transformation, sont :

AU NIVEAU SUPÉRIEUR :

- la dimension k de l'espace de répétition est divisée par la valeur n,

– tous les motifs de la répétition haute auront une dimension de plus, d'une taille de n[2],
– pour tous les tilers d'accès, le vecteur de pavage qui correspond à la dimension k de la répétition est ajoutée dans la matrice d'ajustage,

Observation. S'il s'agit d'un changement de pavage maximal, la dimension k de l'espace de répétition peut être supprimée, en même temps que tous les vecteurs du pavage correspondants ;

[2] Nous avons choisi d'ajouter la dimension au début de la forme multidimensionnelle, observation valable pour le reste de la section.

AU NIVEAU INFÉRIEUR :

Pour chacune des répétition du ce niveau,
– tous les tableaux auront une dimension de plus avec une taille de n,
– pour tous les tilers, une valeur de 0 sera ajoutée dans tous les vecteurs du pavage et d'ajustage, pour s'aligner à l'augmentation des dimensions des tableaux,
– la répétition aura une dimension de plus, toujours avec une taille de n,
– pour tous les tilers d'accès, un vecteur de pavage de forme $\begin{pmatrix} 1 \\ 0 \\ \vdots \end{pmatrix}$ est ajouté à la matrice de pavage.

Définition 13 (Changement de pavage multidimensionnel)**.** Une transformation de changement de pavage par ajout de dimension sur plusieurs dimensions de l'espace de répétition est définie comme la transformation unitaire de plusieurs transformations élémentaires, pour chaque dimension.

Exemple. La Figure 57 montre le résultat de la transformation de changement de pavage (maximal) par ajout de dimensions sur la première dimension du Downscaler après la fusion. La granularité de la répétition du niveau haut n'est plus représentée par des blocs de $(13, 14)$, mais par de 13 lignes de pixels.

Le changement de pavage par ajout de dimension permet l'ajustage de la granularité de la spécification répétitive, mais a aussi deux désavantages majeurs :

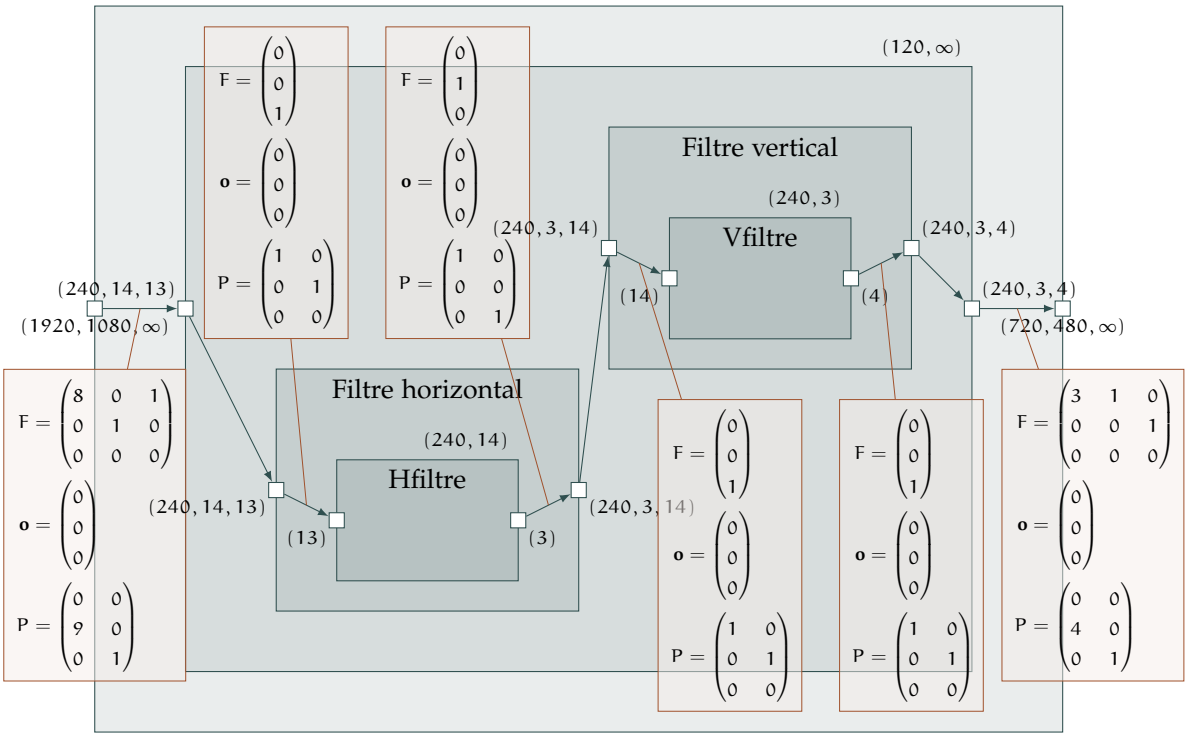
A. ne réduit pas les recalculs ;
B. la segmentation des dimensions : chaque transformation de changement de pavage ajoute des dimensions dans les formes des tableaux et des répétitions du niveau inférieur. Des dimensions initiales peuvent se retrouver partagées en plusieurs morceaux, avec une augmentation de la complexité de la spécification multidimensionnelle.

Plus de dimensions dans la spécification multidimensionnelle compliquent la manipulation des accès et rendent le travail de conception plus laborieux pour l'utilisateur.

Changement de pavage par agrandissement linéaire

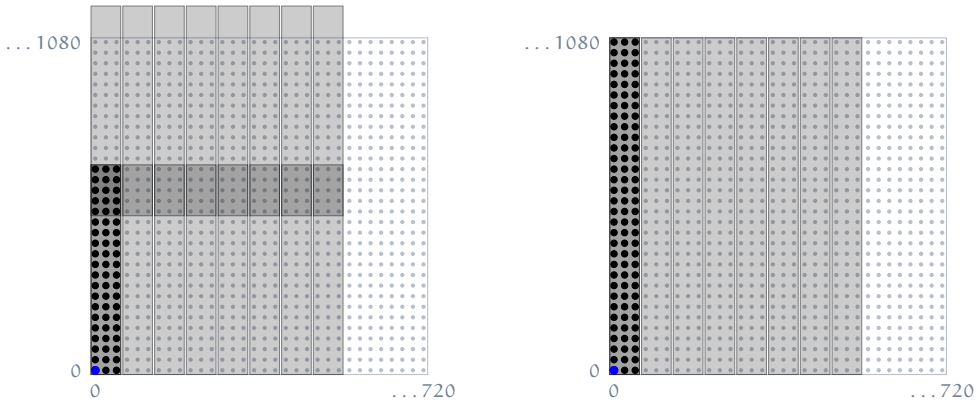
Une transformation conçue dans le but de réduire les recalculs introduits par une fusion est disponible. Cette transformation ressemble au changement de pavage par ajout de dimension, avec la différence que les motifs du niveau supérieur ne sont pas mis un à côté de l'autre dans une dimension additionnelle, mais dans une boîte englobante.

Sur la Figure 58, nous pouvons observer la réduction des recalculs des macromotifs après l'élargissement par la boîte englobante sur la dimension des recouvrements.



La première dimension de l'espace de répétition a descendu un niveau de hiérarchie. Le facteur de changement de pavage, avec une valeur de 240 se retrouve dans les formes des motifs au niveau supérieur de la hiérarchie et dans les formes des tableaux et des répétitions dans le niveau inférieur.

FIG. 57: Le Downscaler après l'ajout de dimensions maximal sur la première dimension



(a) Facteur de 2 (b) Agrandissement maximal

L'agrandissement linéaire avec un facteur de 2 réduit les recalculs, en comparaison avec les chevauchements de la Figure 56 et l'agrandissement maximal sur toute la dimension du glissement réduit complètement les recalculs. Les réductions ont comme effet secondaire une augmentation de la taille du macromotif.

FIG. 58: Réduction des recalculs par agrandissement linéaire

CALCUL DE LA BOÎTE ENGLOBANTE. Toutes les étapes du calcul de l'agrandissement linéaire sont décrites et prouvés dans la thèse de Dumont. Nous n'allons pas rentrer trop dans les détails de calcul, nous limitant aux principes et résultats.

Une transformation d'agrandissement linéaire consiste en plusieurs étapes :

IDENTIFICATION DU RECOUVREMENT et donc des recalculs. Pour qu'il y ait recouvrement, il faut que chaque itération du pavage partage un motif de la sous-tâche avec une autre itération de pavage. Cette condition de traduit dans une équation

$$\overrightarrow{P_{op_i}} - \overrightarrow{\mathcal{MF}_{op_k}} \times \Delta \equiv 0 \mod \overrightarrow{M} \tag{5.16}$$

qui dit simplement qu'une origine de motif originel soit atteignable par une dimension de macroajustage dans un premier macromotif $\overrightarrow{\mathcal{MF}_{op_k}} \times d_k$ et que cette même origine soit également atteignable par la même dimension de macroajustage $\overrightarrow{\mathcal{MF}_{op_k}} \times d'_k$ après une itération de pavage sur le vecteur $\overrightarrow{P_{op_i}}$. Dans la pratique, la condition se traduit dans un algorithme[3] capable à fournir tous les couples (i, k, Δ) avec les dimensions où il y a de recouvrement ;

[3] Disponible dans la thèse de Dumont.

CHOIX DE LA TRANSFORMATION entre les alternatives disponibles. Pour chaque couple (i, k, Δ) et un facteur de changement de pavage de n on peut calculer un taux de recouvrement égal à

$$\frac{\mathcal{MD}_{op_k} + (n-1)|\Delta|}{n|\Delta|} \tag{5.17}$$

qui peut guider le choix de la transformation ;

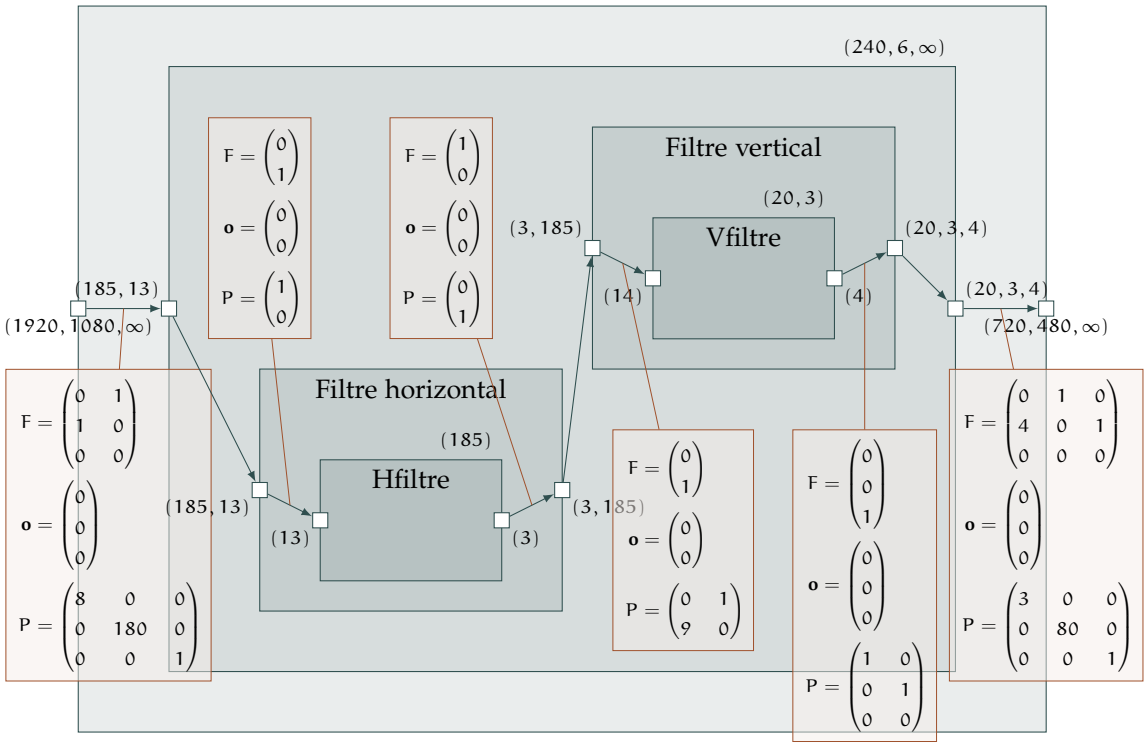
APPLICATION DE LA TRANSFORMATION Une fois la transformation et le facteur d'agrandissement choisi, une succession d'opérateurs ODT sont appliquées, permettant le calcul des nouvelles valeurs pour les répétitions (tailles de tableaux, motifs, tilers d'accès) des deux niveaux de la hiérarchie.

Exemple. La Figure 59 montre le résultat de la transformation de changement de pavage par agrandissement linéaire sur la dimension du glissement avec un facteur de 20 qui réduit les recalcule à un facteur de $185 \times 6/1080 = 1.02(6)$. Pour les calculs, identifier une transformation d'agrandissement linéaire se résume à identifier les indices i, k et le déplacement Δ de l'Équation 5.16, pour $P_{op} = \begin{pmatrix} 8 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, $\mathcal{MF}_{op} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}$ et $|\Delta| < \mathcal{MD}_{op_k} = \begin{pmatrix} 14, 13 \end{pmatrix}_k$. La solution est représentée par le couple $(i, k, \Delta) = (1, 0, 9)$: la première dimension du macroajustage[4] est atteignable par une itération du pavage sur la deuxième dimension. Une transformation d'agrandissement linéaire pour cette solution avec un facteur de $n = 20$ réduit les recalculs, en utilisant l'Équation 5.17, à une valeur de $\frac{14+(20-1)\times 9}{20\times 9} = \frac{185}{180} = 1.02(6)$, la même valeur que celle calculée par la division des répétitions sur la figure.

[4] Les indices des dimensions commencent à 0.

5.3.3 *Aplatissement*

Définition 14 (Aplatissement)**.** Nous avons vu comment l'utilisation d'un changement de pavage maximal sur une dimension de l'espace de répétition fait que cette dimension descend un niveau de la hiérarchie. Un changement de pavage maximal sur toutes les dimensions de la répétition du niveau supérieur aura comme effet l'élimination de la répétition entière au niveau supérieur. Le niveau supérieur de la hiérarchie est donc inutile et sera supprimé de la spécification, en le



Un agrandissement linéaire avec un facteur de 20 sur la dimension du recalcul réduits les répétitions de la première tâche d'un facteur de ≈ 1.56 à un facteur de ≈ 1.03, mais avec une augmentation de la taille du macromotif avec un facteur de ≈ 13.21, soit 513 éléments mémoire de plus.

FIG. 59: Modèle Downscaler après réduction des recalculs

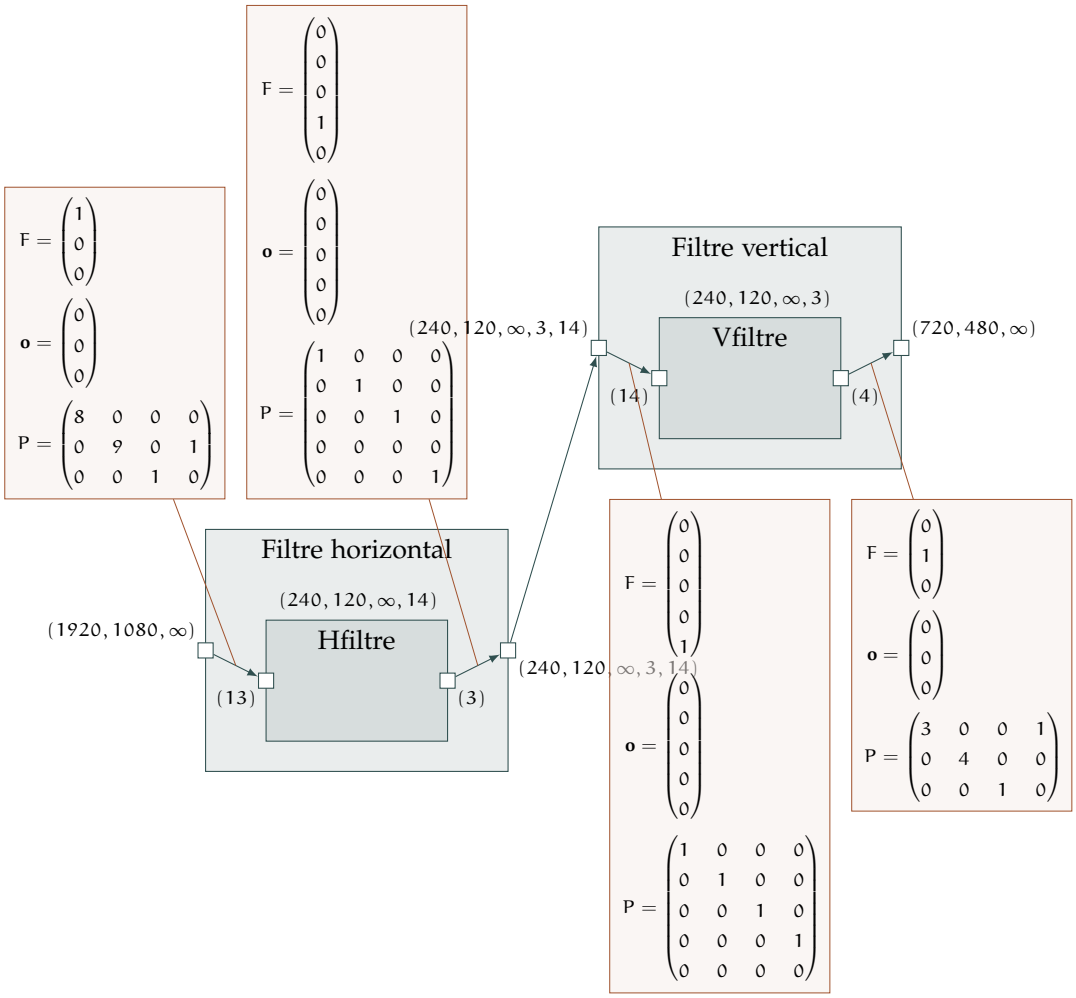
remplaçant par le niveau inférieur de répétitions. C'est ce qu'on appelle *aplatissement*[5].

[5] *Collapse* en anglais.

Observation. Au moment de la suppression du niveau supérieur de la hiérarchie, même s'il n'y a pas de répétition, les tilers doivent être composés avec les tilers correspondants du niveau inférieur par le court-circuit (*cf* l'Équation 5.2.3) pour faire le lien direct entre les tableaux du niveau supérieur et les motifs du niveau inférieur.

Exemple. La Figure 60 montre le résultat de l'aplatissement sur le modèle du Downscaler après la fusion. Cette nouvelle spécification est équivalente au celle d'avant la fusion, mais avec de motifs d'accès différents. La taille du tableau intermédiaire est aussi modifiée et l'utilisation du changement de pavage par ajout de dimension fait que les recalculs n'ont pas disparu. La construction de court-circuit permet l'expression directe des accès aux motifs du deuxième niveau de la hiérarchie dans les tableaux du niveau haut. Prenant par exemple le tiler d'entrée, sur la Figure 55, l'accès aux tableaux $(1920, 1080, \infty)$ par des motifs de taille (13) sans passer par les macromotifs de taille $(13, 14)$ se fait par le tiler calculé avec les relations de court-circuit, comme dans le calcul suivant :

$$O = O_{up} + F_{up}.O_{sub} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}.\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$



Sans l'utilisation du court-circuit pour les deux tilers du niveau supérieur, on se serait retrouvé avec des tilers similaires aux tilers d'accès au tableau intermédiaire aussi pour les tilers extérieurs. Même s'il n'y a pas de répétition, un tiler exprime le placement d'un motif dans le tableau, mais qui occupe toute la taille du tableau. Dans le cas du tiler d'entrée, le placement d'un motif de $(240, 120, \infty, 13, 14)$ dans le tableau de taille $(1920, 1080, \infty)$.

FIG. 60: Downscaler après l'aplatissement

$$P = \begin{pmatrix} P_{up} & F_{up}.P_{sub} \end{pmatrix} = \left(\begin{pmatrix} 8 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} . \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} 8 & 0 & 0 & 0 \\ 0 & 9 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$F = F_{up}.F_{sub} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} . \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} .$$

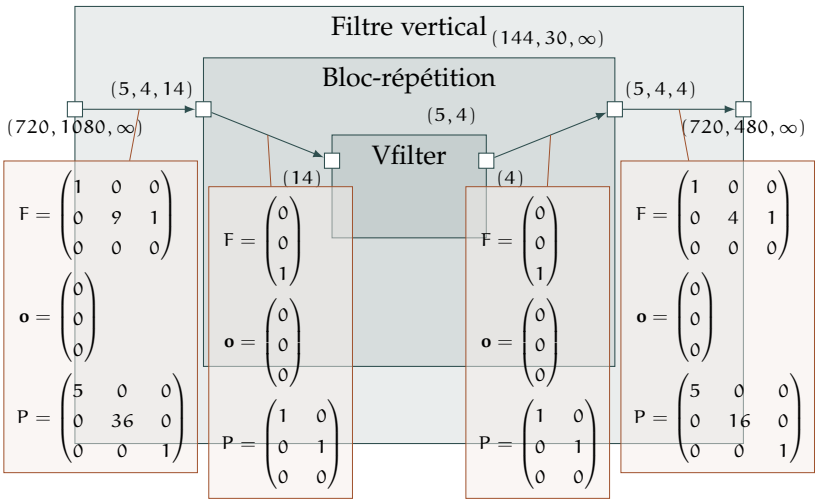
Ce tiler est exactement le tiler d'entrée après l'aplatissement.

5.3.4 *Tiling*

La transformation d'aplatissement, comme nous venons de voir, permet la suppression d'un niveau de hiérarchie. Une transformation capable de faire l'opération inverse est disponible : la création d'un

niveau de hiérarchie. Dumont dans sa thèse présente la transformation de tiling qui seulement ajoute un niveau de la hiérarchie sans répétition au niveau inférieur. Même si une transformation de changement de pavage ultérieure pourrait déplacer des répétitions du niveau supérieur à celui inférieur, nous préférons faire les deux opérations en même temps : au moment de la création de la hiérarchie, l'espace de répétition est partagé en blocs et les répétitions des blocs seront au niveau inférieur, pendant que les répétitions des blocs resteront au niveau supérieur.

Exemple. Dans la description des dépendances interrépétition connectées à travers la hiérarchie (sous-section 2.6.4) nous avons vu le résultat d'une transformation de tiling sur un sous-ensemble du filtre vertical avec des dépendances interrépétition. La Figure 61 montre le partage en blocs de $(5,4)$ sur le filtre vertical de la Figure 54. Le résultat de la transformation est similaire avec la transformation de changement de pavage par ajout de dimension si on considérait un niveau inférieur de hiérarchie avec une répétition vide, $()$.



L'espace de répétition a été partagé en $(144, 30, \infty)$ blocs de $(5, 4)$.

FIG. 61: Filtre vertical après une transformation de *tiling*

Cette transformation peut être utilisée pour changer la granularité d'une répétition, la création de la hiérarchie quand une fusion n'est pas applicable et permet d'augmenter la localité des données au niveau inférieur de la hiérarchie.

5.3.5 *Conclusions sur l'utilisation des transformations*

Pour résumer, les transformations de haut niveau ARRAY-OL sont des transformations data-parallèles au niveau de la spécification basées sur le formalisme ODT pour exprimer les dépendances et garantir la validité des opérations.

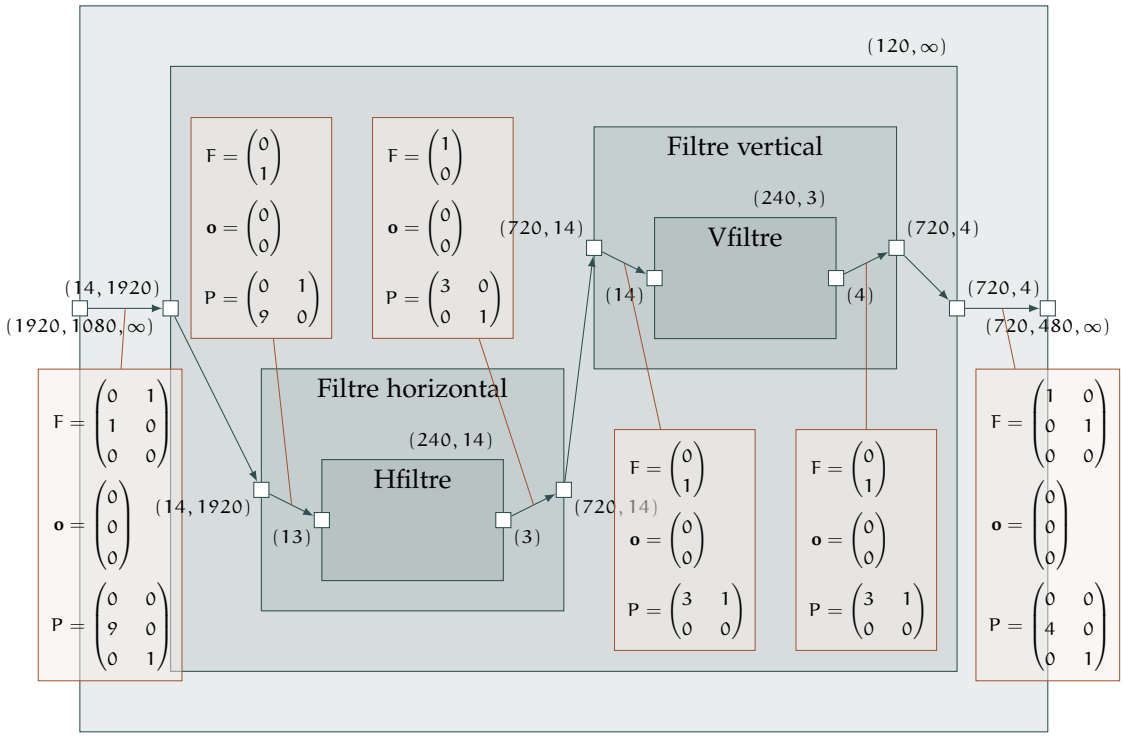
Les transformations permettent le refactoring de la spécification pour l'adapter au passage à un modèle d'exécution, en distribuant les répétitions data-parallèles à travers la hiérarchie et ainsi permettant le choix d'un passage direct à l'exécution.

Les différentes transformations peuvent être utilisées pour optimiser la spécification, par :

- la réduction des tableaux intermédiaires ;
- le changement du degré du parallélisme ;
- la réduction des recalculs ;
- augmenter la localité des données ;
- adapter la spécification aux contraintes d'exécution.

ADAPTER LA SPÉCIFICATION AUX CONTRAINTES D'EXÉCUTION.

Nous avons vu comment utiliser les transformationsARRAY-OL pour prendre en compte les contraintes internes : comment enchaîner les calculs. Des contraintes d'exécution peuvent aussi influencer le choix du refactoring. Sur l'application du Downscaler, une telle contrainte peut spécifier par exemple que les images arrivent pixel par pixel sur la dimension horizontale. L'extension des macromotifs au niveau haut pour inclure de lignes entières de données, en utilisant une transformation d'agrandissement linéaire maximale sur la dimension horizontale, permet de mieux prendre en compte cette contrainte et une meilleure gestion des tampons de données. Le résultat est montré sur la Figure 62.



Le niveau supérieur de la hiérarchie prend des tuiles contenant des lignes entières des images. Moins de parallélisme est exprimé à ce niveau, mais vu que les images arrivent ligne par ligne, le mécanisme de tampons est simplifié.

FIG. 62: Modèle Downscaler qui respecte les contraintes de flot de données

5.4 EXTENSIONS

Les transformations de haut niveau ARRAY-OL, leur définition et formalisation utilisant les ODT ont été au cœur des thèses de Soula

et Dumont. Des besoins pratiques nous ont conduits à étendre ces transformations. Deux directions d'extension peuvent être notées :

A. le regroupement de plusieurs transformations élémentaires dans une seule transformation unitaire ; cela facilite l'utilisation des transformations et permet de remonter des propriétés assez intéressantes ;

B. l'extension des transformations pour des cas non traités où traités dans une manière non appropriée.

Nous avons déjà fait référence dans la présentation des transformations aux extensions possibles. Pour récapituler, les groupements de plusieurs transformations unitaires mentionnées sont :

CHANGEMENT DE PAVAGE MULTIDIMENSIONNEL. Une transformation unitaire constituée de plusieurs changements de pavages sur des dimensions différentes de l'espace de répétition ;

CHANGEMENT DE PAVAGE MAXIMAL. Une transformation de changement de pavage avec un facteur de changement de pavage égal à la taille de la dimension de l'espace de répétition concernée ; la dimension entière descend un niveau de la hiérarchie ;

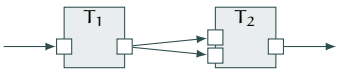
PARTAGE D'UN ESPACE DE RÉPÉTITION EN BLOCS. C'est ce qu'on appelle tiling, mais en fait représente deux opérations distinctes, une opération de tiling comme celle décrite par Dumont pour créer un niveau de hiérarchie et une transformation de changement de pavage par ajout de dimensions ;

FUSIONS MULTIPLES. La fusion de plusieurs répétitions qui se succèdent, en utilisant plusieurs fusions élémentaires suivies par des transformations d'aplatissement pour limiter à deux le nombre des niveaux de hiérarchie résultat.

Comme extension des transformations, le plus de restrictions sont définies sur la fusion élémentaire, dans le cas de plusieurs tableaux intermédiaires. Comme extension on mentionne :

AGRANDISSEMENT LINÉAIRE SANS RECALCULS. La transformation d'agrandissement linéaire est conçue pour réduire les recalculs et la première étape est exactement la détection des recalculs. Une telle transformation est intéressante même dans les cas où il n'y a pas de recalculs. Nous proposons d'étendre la transformation pour des motifs qui ne se chevauchent pas, mais qui sont « collés » un après l'autre ;

TABLEAU INTERMÉDIAIRE CONSOMMÉ PLUSIEURS FOIS. C'est le cas d'un tableau intermédiaire qui est consommé par la deuxième tâche en deux motifs d'accès différents :



ÉLIMINATION DE LA MULTIPLICITÉ INUTILE. Des transformations de refactoring peuvent amener à des structures multidimensionnelles contenant des dimensions de taille unitaire. Pour simplifier la description multidimensionnelle, ces dimensions peuvent et devraient être éliminés. Cette simplification implique aussi :
- dans le cas de la forme d'une répétition, la suppression des vecteurs de pavage qui correspondent à la dimension éliminé,
- dans le cas de la forme d'un tableau, la suppression récursive des dimensions dans les tableaux connectés à celui-ci, la suppression des lignes dans les matrices d'ajustage et de pavage

dans les tilers qui accèdent le tableau et des vecteurs d'ajustage si le tableau exprime la forme du motif dans une description répétitive.

Nous allons nous arrêter seulement sur les extensions plus complexes dont nous n'avons pas discuté dans la présentation des transformations.

5.4.1 *Agrandissement linéaire sans recalculs*

L'agrandissement linéaire identifie les dimensions où des recouvrements de motifs se produisent et donc des recalculs. En augmentant la taille du motif sur cette dimension en prenant la boîte englobante pour contenir plusieurs motifs initiaux, la quantité de recalculs est diminuée.

L'utilisation de la même transformation (même quand il n'y a pas des recalculs) au détriment du changement de pavage par ajout de dimension, a ces avantages :

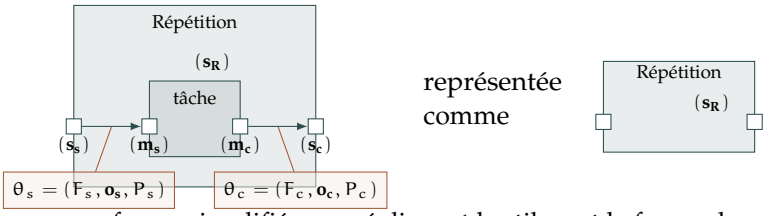
- on peut éviter la segmentation des dimensions et l'apparition des tableaux multidimensionnels avec excessivement de dimensions ;
- gérer les tampons de données pour des dimensions linéarités est plus simple et plus directe.
  Exemple. Le modèle du Downscaler qui respecte les contraintes du flot de données sur la Figure 62 est le résultat d'une telle transformation. Sur la dimension horizontale, il n'y a pas de recalculs et donc la transformation d'agrandissement linéaire n'est pas permise sans cette extension.

Au niveau de la formalisation de la transformation, la modification se retrouve au niveau de l'algorithme qui cherche les dimensions du recalcul. La condition de recalcul est modifiée pour retourner les dimensions où les motifs sont collés un après l'autre.

5.4.2 *Fusions multiples*

Une fusion élémentaire calcule la répétition commune des deux répétitions qui se succèdent, en minimisant les tableaux intermédiaires produits par la première et consommés par la seconde. Dans des applications réelles, plusieurs répétitions peuvent se succéder et l'objectif est de réduire si possible tous les tableaux intermédiaires entre ces répétitions. Comme nous allons voir, cet objectif dépend de plusieurs paramètres et n'est pas facilement atteignable.

Si on considérait une répétition



sous une forme simplifiée, en négligeant les tilers et la forme des motifs, une succession de répétitions à un niveau de hiérarchie ressemblait à la Figure 63[6].

Dans une telle configuration, le but est de réduire toutes les tailles des tableaux intermédiaires. Cette opération est possible par l'enchaînement de plusieurs fusions élémentaires pour calculer une répétition commune pour chaque couple *répétition productrice* $\oplus$[7] *répétition consommatrice*. L'algorithme de fusion doit suivre les règles suivantes :

[6] Nous nous intéressons toujours aux fusions avec un seul tableau intermédiaire entre chaque deux répétitions.

[7] $\oplus$ = opération de fusion.

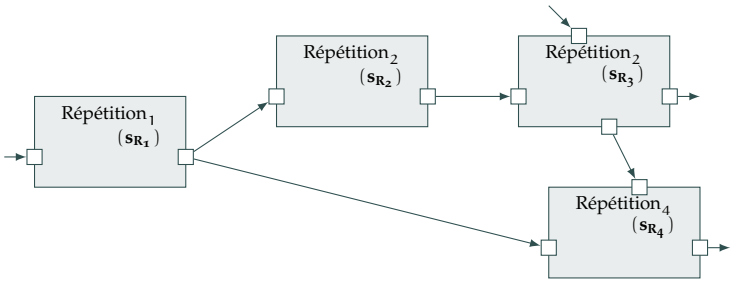


FIG. 63: Enchaînement réel de répétitions

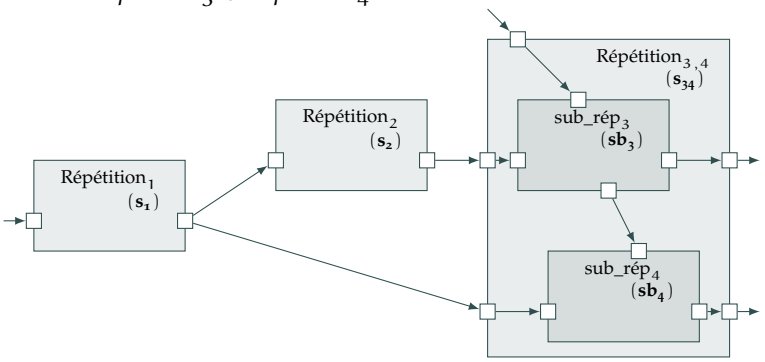
- on commence avec la fusion de deux répétitions ;
- à chaque étape, le résultat de la fusion antérieure est fusionné avec une nouvelle répétition ;
- l'enchaînement des répétitions est imposé par les tableaux intermédiaires entre les répétitions ; dans une fusion, la première répétition doit produire un tableau consommé par la deuxième. Á la fin l'enchaînement est dicté par l'ordre partiel entre les répétitions.
  Exemple. Sur la structure de la Figure 63, le seul ordre de fusion possible est : $Répétition_1 \oplus Répétition_2 \oplus Répétition_3 \oplus Répétition_4$;
- pour limiter à deux les niveaux de hiérarchie résultat, chaque fusion (exceptant la première) est suivie par une opération d'aplatissement pour éliminer le niveau supplémentaire de hiérarchie.

Observation. L'ordre partiel entre les répétitions détermine l'ordre des fusions, mais la fusion est une opération non-associative et la façon de faire cette association influence les résultats. L'ordre de la structure sur la Figure 63 peur être interprété comme :

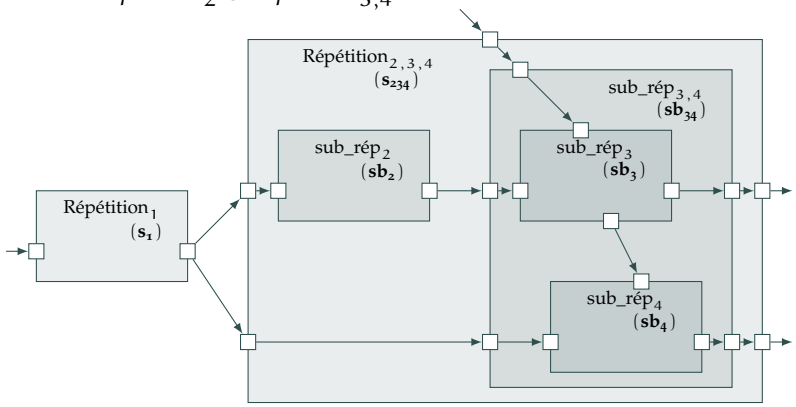
- $(Répétition_1 \oplus (Répétition_2 \oplus (Répétition_3 \oplus Répétition_4)))$;
- $(((Répétition_1 \oplus Répétition_2) \oplus Répétition_3) \oplus Répétition_4)$;
- $(Répétition_1 \oplus ((Répétition_2 \oplus Répétition_3) \oplus Répétition_4))$;
- etc.

Exemple. Pour la structure de la Figure 63, les étapes de la fusion multiple, si on choisi l'association $(Répétition_1 \oplus (Répétition_2 \oplus (Répétition_3 \oplus Répétition_4)))$, sont :

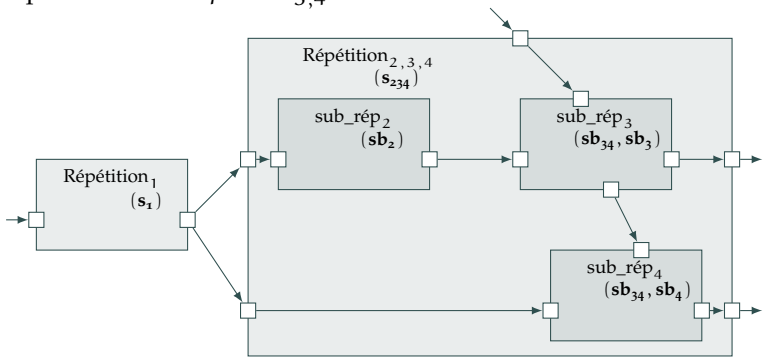
1. Fusion $Répétition_3 \oplus Répétition_4$:



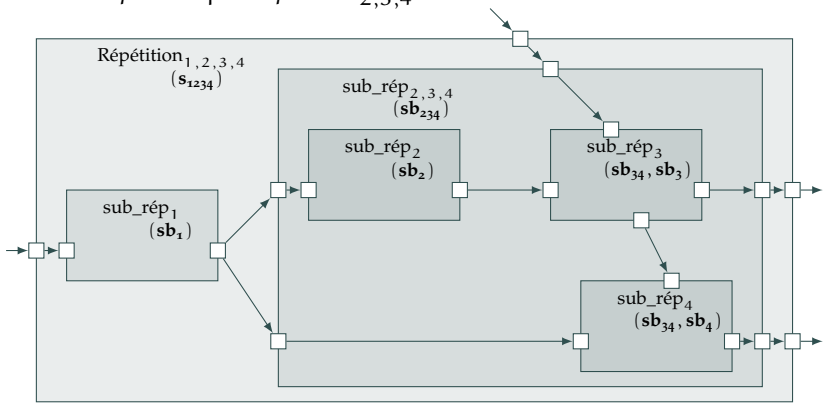
2. Fusion $\textit{Répétition}_2 \oplus \textit{Répétition}_{3,4}$:



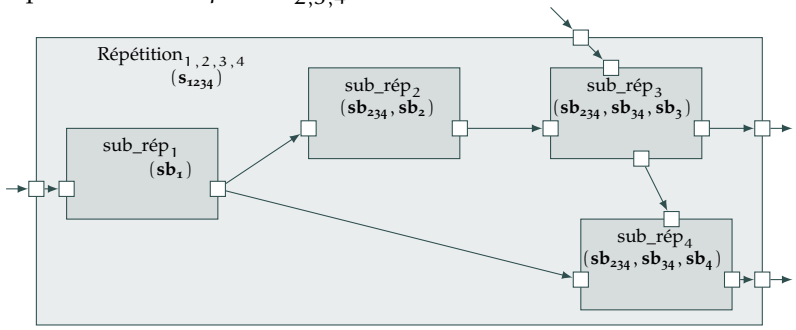
3. Aplatissement $\textit{Répétition}_{3,4}$:



4. Fusion $\textit{Répétition}_1 \oplus \textit{Répétition}_{2,3,4}$:



5. Aplatissement $\textit{Répétition}_{2,3,4}$:



Observation. Par une fusion multiple, seulement le dernier[8] tableau intermédiaire est minimisé. Les autres tableaux intermédiaires représentent des groupes minimaux de motifs nécessaires pour l'exécution au moins une fois de la dernière sous-répétition, et non de la sous répétition qui consomme le tableau respectif.

[8] Selon l'ordre de la fusion.

Le choix de l'ordre de fusion influence les résultats d'une fusion multiple. Dans le contexte des optimisations en taille mémoire en utilisant des fusions, cet ordre est essentiel. Dans le chapitre 7, nous allons étudier ce problème plus en détail sur une application réelle de traitement de signal.

5.4.3 *Tableau intermédiaire consommé plusieurs fois*

Dans la présentation de la transformation de fusion de la sous-section 5.3.1, nous avons vu comment les cas avec plus de tableaux produits ou consommés sont traités. La plupart de cas peuvent être réduits aux calculs successifs de la même fusion avec différentes parties opérandes de la première tâche ou parties résultat de la seconde.

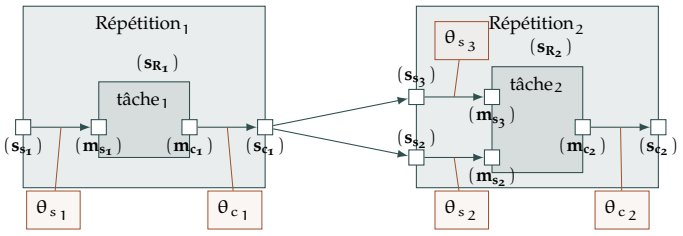
Le vrai problème se pose dans le cas de plusieurs tableaux intermédiaires. La représentation sous la forme ODT complète ne fonctionne pas ; cette représentation bloque le calcul de la fusion à cause d'une inversion interdite de matrice. La solution proposée par Dumont consiste à copier la première répétition autant de fois qu'il y a de tableaux intermédiaires et à calculer la fusion avec la deuxième en considérant un tableau à la fois. Ce choix a des inconvénients assez importantes :

- la création d'un nombre de niveaux de hiérarchie égal au nombre de tableaux intermédiaires, qui peuvent amener aux hiérarchies abyssales ;
- les tableaux intermédiaires ne sont pas réduits en même temps ;
- les recalculs introduits par la duplication de la première répétition.

Tous ces motifs nous ont conduit à chercher des alternatives pour ce type de transformation, au moins pour des cas spéciaux. C'est le cas de la consommation multiple d'un tableau intermédiaire par la deuxième répétition. Cela peut sembler improbable, mais dans des applications de traitement intensif de signal ce type de consommation est assez fréquente, même si elle est cachée. C'est le cas de l'exemple présenté sur la Figure 63, où même si sur la figure on ne retrouve pas directement une telle consommation, on peut voir qu'entre la $\mathit{Répétition}_1$ et $\mathit{Répétition}_4$ il y a deux chemins de données. Dans le cas d'une fusion multiple de toutes ces répétitions on va de retrouver à un moment ou l'autre avec une consommation multiple entre deux répétitions.

Exemple. Dans le cas de la fusion multiple présentée précédemment, la 5^{e} étape (fusion $\mathit{Répétition}_1 \oplus \mathit{Répétition}_{2,3,4}$) représente une fusion avec un tableau intermédiaire consommé plusieurs fois.

PROPOSITION. Une consommation multiple d'un tableau intermédiaire par la deuxième répétition à une structure de type :



La fusion des deux répétitions devrait réduire les tableaux intermédiaires en calculant une répétition commune. Dans ce cas nous avons un seul tableau intermédiaire, mais consommé plusieurs fois avec des motifs d'accès différents. Pour réduire le tableau intermédiaire, il faut

calculer un macromotif qui englobe les deux macromotifs correspondant à chaque consommation.

Pour trouver ce macromotif global, nous proposons de calculer un motif commun opérant pour la deuxième répétition et d'appliquer la transformation de fusion considérant un seul tableau intermédiaire qui est consommé par ce motif commun par la deuxième répétition. C'est comme si on ajoutait un niveau de hiérarchie pour la deuxième répétition, illustré sur la Figure 64.

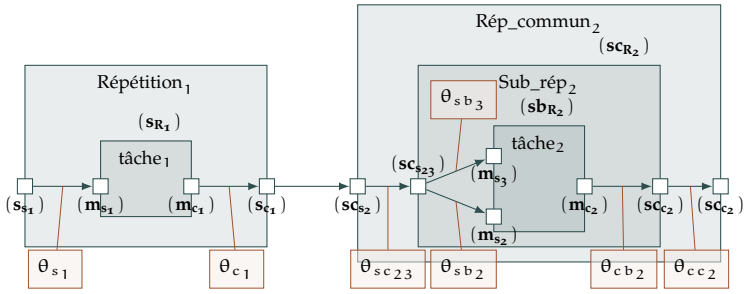
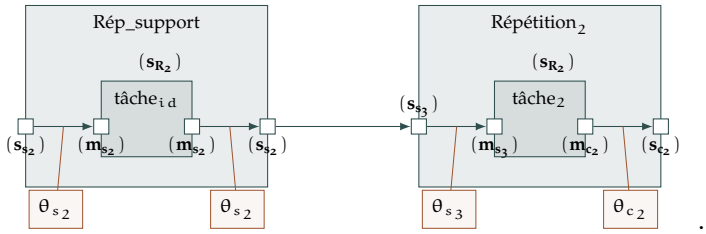


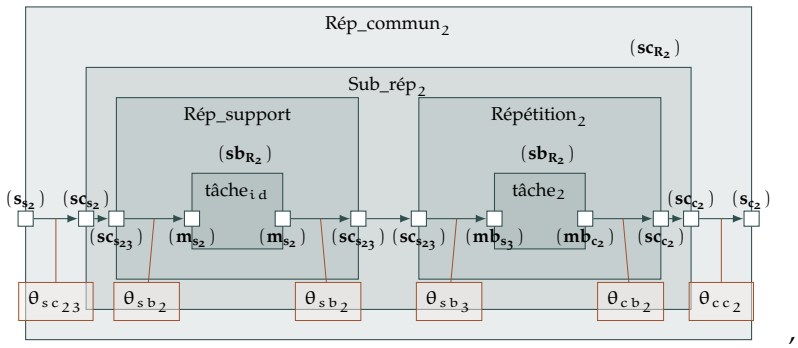
FIG. 64: Motif commun pour la consommation multiple.

Le calcul du motif commun opérande peut se faire en se servant d'une construction support et l'application d'une transformation de fusion élémentaire pour calculer le motif commun minimal. La construction support est réalisée par la séparation d'un tiler d'accès dans une autre répétition qui sera la première répétition de la fusion :



.

Suite à la fusion de la répétition support avec la deuxième répétition sans la consommation du tableau intermédiaire utilisé pour la création de la répétition support,



,

on se retrouve avec le motif commun minimal pour les deux consommations et les tilers d'accès de la Figure 64.

Toutes les étapes de la fusion doivent être transparentes pour l'utilisateur.

Après le calcul du motif commun pour les consommations de la deuxième répétition, une transformation de fusion élémentaire est utilisée pour réduire le tableau intermédiaire entre les deux répétitions. Une

transformation d'aplatissement est utilisée par la suite pour éliminer le niveau de hiérarchie introduit par la fusion pour le calcul du motif commun consommé.

Observation. Nous avons présenté le rationnement de la fusion pour le cas de deux consommations dans la deuxième répétition. Dans le cas de plus de deux consommations l'algorithme reste le même, mais il faut plusieurs étapes pour le calcul du motif consommé commun : pour chaque consommation nous avons besoin d'une fusion avec une répétition support, suivi par des opérations d'aplatissement derrière pour limiter le nombre de niveaux de hiérarchie.

5.4.4 *Résumé*

Nous avons présenté les transformation data-parallèles disponibles pour le refactoring de haut-niveau et nous avons proposé une série d'extensions, intéressantes surtout dans une perspective plus pratique.

Par la suite nous allons essayer de tirer quelques associations entre ces transformation de haut niveau et les transformations de boucles.

5.5 COMPARAISON AVEC LES TRANSFORMATIONS DE BOUCLES

Comme nous avons discuté dans le chapitre 4, les techniques d'optimisation employant les transformations de boucles visent l'amélioration de la régularité et la localité d'accès aux données et la suppression/réduction des tampons du niveau système du code d'une application.

Nous avons vu que ces transformations de code sont plus efficaces sur un code de traitement de données extrêmement régulier : des nids de boucles parfaits. Cela est précisément le domaine d'applications visé par Array-OL et, comme nous avons vu, la traduction vers une description basée sur boucles est directe.

Les ressemblances évidentes entre les deux types de transformation nous ont amenés à essayer d'identifier les connexions entre les deux domaines et d'enquêter sur les optimisations techniques basées sur des transformations de boucles et si des correspondances possibles avec les optimisations en Array-OL.

Nous commençons avec quelques observations importantes sur les deux types de transformations :

- contrairement aux transformations de boucles qui sont des usuellement des optimisations locales, les transformations Array-OL sont des transformations qui peuvent être utilisées à n'importe quel niveau de la hiérarchie grâce à l'accès aux données basé sur des motifs uniformes ;
- les structures d'accès Array-OL sont rendues plus visibles et plus facilement utilisables par les accès uniformes, contrairement aux représentations complexes des indices de boucles.

Il y a bien évidemment de désavantages dans l'utilisation d'Array-OL :

- la régularité restreint le domaine d'application à un ensemble relativement limité ;
- les outils d'optimisation sont, pour le moment, semi-automatiques, des transformations de refactoring dans les mains du concepteur.

La complexité des algorithmes d'optimisation force l'introduction des modalités pour représenter le problème (contraintes, transformation,

fonctions de coût) par l'utilisation de formalismes plus efficaces qui pourront faciliter la manipulation des concepts tels que : conformité, dépendances de donnés, objectifs. L'introduction d'un formalisme est extrêmement importante pour la partie décision de l'optimisation. Des algorithmes corrects et complexes ont besoin d'être conçus autour de tels formalismes. La spécification ARRAY-OL est conçue autour d'un tel formalisme et aussi les transformations sont spécifiées et leurs conformités est prouvée par l'utilisation du formalisme ODT basé sur l'algèbre linéaire.

Les noms des transformations ARRAY-OL *ont été choisis pour montrer les similarités avec les transformations de boucles homonymes.*

Nous n'allons pas faire une comparaison séparée pour chaque paire de transformations, chaque transformation ARRAY-OL a une fonctionnalité similaire à son homonyme, mais plutôt essayer d'identifier le rôle de chaque transformation et son usage potentiel.

Au moment du passage à un modèle d'exécution à partir une spécification ARRAY-OL, un ensemble d'éléments clefs doit être attentivement analysé. Tout d'abord, nous devons isoler les dimensions infinies, respecter les contraintes internes introduites par les dépendances de données et en même temps éviter les points de blocage dans l'exécution.

Pour tout cela on peut utiliser la transformation de fusion qui a comme effets : l'isolation des dimensions infinies au niveau haut de la hiérarchie, la minimisation des tableaux intermédiaires et la garantie d'une exécution sans blocage.

Comme pour la fusion des boucles, les deux ont le rôle d'unifier deux entités dépendantes (des répétitions dans le cas d'ARRAY-OL et des nids de boucles dans l'autre cas). La fusion de boucles est une transformation de programme qui combine plusieurs boucles en une seule et elle est utilisée dans les compilateurs-parallèliseurs principalement pour augmenter la granularité des boucles et pour améliorer la réutilisation des données.

Néanmoins, la fusion de boucles englobe d'optimisations beaucoup plus complexes et peut être utilisée pour attendre diffèrent objectifs en collaboration avec d'autres transformations de boucles : réduire le coût de l'évaluation des limites des boucles, améliorer la localité temporelle des données, réduction des synchronisations quand les boucles sont distribuées sur plusieurs unités d'exécution. La fusion de boucles peut avoir un impact indirect sur les performances en autorisant d'autres optimisations très utiles qui sont limitées à de blocs de base ou de boucles parfaites.

La fusion ARRAY-OL est conçue dans le but de réduire la taille mémoire nécessaire pour le stockage des données intermédiaires. Dans ce sens, elle a des similitudes avec la technique d'optimisation de boucles qui utilise la fusion et des autres transformations pour réduire les tailles des tableaux intermédiaires telles que le remplacement scalaire ou l'optimisation de l'ordre de stockage intra/inter-tableau.

De Greef *et al.* présentent en [25] une stratégie qui permet la réduction des tailles mémoires nécessaires pour des applications multimédia data-intensives visant la réutilisation efficiente des emplacements mémoires par d'optimisations de l'ordre de stockage. Premièrement, chaque tableau est optimisé indépendamment par la réduction à sa fenêtre de référence (optimisations intra-tableau) pour augmenter la réutilisation des emplacements mémoires pour les éléments du même tableau. Après une deuxième phase, différents tableaux peuvent partager des placements en mémoire si leur durée de vie ne se chevauche pas (opti-

misations inter-tableau). La fusion des répétitions Array-OL permet dans la même façon de calculer la fenêtre de référence minimale entre la production/consomption des tableaux intermédiaires.

La transformation de fusion peut être utilisée pour restructurer l'application vers une spécification hiérarchique spéciale où toutes les dimensions infinies sont isolées à un niveau haut de la hiérarchie qui va représenter le niveau du flot-données.

Une différence entre la fusion Array-OL et les transformations de boucles est l'introduction de recalculs dans la spécification. Sans l'introduction des recalculs pour des accès chevauchants, la réduction des tableaux intermédiaires en gardant une répétition commune data-parallèle n'est pas possible en Array-OL. Les recalculs sont des compromis entre le parallélisme et la réduction des tailles mémoires.

La transformation d'aplatissement a un rôle important en connexion avec la fusion, en évitant l'apparition des « hiérarchies abyssales », crées par le chaînage de fusions élémentaires.

Le changement de pavage ressemble au déroulement de boucles. Les deux transformations agissent sur la redistribution des itérations entre les niveaux (de la hiérarchie ou des nids de boucles). Dans le contexte d'Array-OL, cette transformation peut aussi être utilisée pour restructurer la spécification pour respecter les contraintes d'exécution ou d'environnement.

La transformation Array-OL de tiling corresponde au partage de boucles ; la première introduit un niveau de hiérarchie pendant que la deuxième un niveau de nidification dans le nid de boucles. Les deux ont le rôle de partager l'espace d'itération en blocs fonctionnels pour augmenter la localité des données.

Nous devrons noter que dans le contexte des optimisations Array-OL nous ne sommes pas intéressés d'augmenter le parallélisme dans l'application. Tout le parallélisme est disponible par construction ; le but d'Array-OL est de fournir un langage de modélisation de haut niveau où le maximum de parallélisme est entièrement disponible au niveau de la spécification.

Nous sommes plus intéressés dans des optimisations mémoires (statique et dynamique), en respectant les contraintes de la spécification. Néanmoins, les transformations changent la structure de la spécification et cela provoque des changements dans le parallélisme (ou son organisation).

Considérant les similitudes entre les deux types de transformations, nous pouvons essayer de prendre des résultats des techniques d'optimisation des boucles, dont on peut dire qu'elles ont atteint le niveau de maturité dû aux études extensives du sujet, et utiliser ces résultats dans le contexte d'Array-OL.

Des problèmes de complexité font que les algorithmes capables de donner une solution optimale pour des optimisations mémoires ne sont pas pratiques. On utilise le plus souvent des heuristiques dont l'efficacité est prouvée. C'est dans cette direction que nous avons orienté nos études sur des optimisations basées sur les transformations de haut niveau Array-OL.

Dans cette section nous avons essayé de tracer des parallèles entre les transformationsArray-OL et les transformations de boucles. Le cœur du langage Array-OL est la représentation visuelle de boucles data-parallèles (ce que nous appelons répétitions) et les transformations Array-OL sont conçus pour manipuler ces répétitions, de la même

manière que les transformations de boucles manipulent les boucles. Même si les transformations ARRAY-OL peuvent être considérés comme des transformations de boucles de haut niveau et les deux types de transformations partagent des similitudes, les deux contextes différents où elles sont utilisées changent radicalement leur rôle.

Les transformations de boucles sont utilisées au niveau de la compilation du code et leur utilisation dépend des stratégies d'optimisation implémentées dans le compilateur et en grande partie par la plateforme d'exécution ciblée. Au contraire, les transformationsARRAY-OL sont au niveau haut de la spécification et sont indépendantes du code final généré, même si elles peuvent aussi être dirigées par certaines caractéristiques de la plateforme cible.

Les transformations ARRAY-OL ont été conçus pour permettre le refactoring d'une spécification ARRAY-OL à un niveau haut de spécification visuelle, en manipulant les concepts de répétition, hiérarchie, granularité. La contrainte que le résultat d'une telle transformation reste au même niveau d'abstraction de la spécification ARRAY-OL (décomposition hiérarchique en répétitions data-parallèles avec des accès uniformes) limite l'éventail de transformations permises.

Les deux types de transformations restent aux deux niveaux différents d'abstraction et elles peuvent être complémentaires, les transformations de haut niveau pour la manipulation de la structure globale de l'application et des transformations de boucles au niveau de compilation du code généré, pour des optimisations de bas niveau.

5.6 CONCLUSIONS

Dans le chapitre 7 nous allons présenter une stratégie d'optimisation par l'utilisation des transformations de boucle, illustrée sur une application réelle de traitement intensif de signal.

Avant de passer à la validation pratique des résultats théorétiques autour les transformations de boucle, nous devrons analyser l'interaction de ces transformation avec les dépendances uniformes.

Les dépendances interrépétition n'existaient pas au moment de la conception du formalisme ODT et des transformations de haut-niveau data-parallèles et ainsi une représentation en ODT d'une répétition n'exprime pas les éventuelles dépendances uniformes entre les répétitions. Pour pouvoir en même temps exprimer ce type de dépendances et pouvoir profiter des transformations de refactoring, l'interaction entre les deux doit être gérée.

Une possibilité sur laquelle nous avons enquêté a été le choix d'étendre la représentation en ODT d'une répétition pour l'expression des dépendances uniformes (avec l'extension des opérateurs ODT si nécessaire) et de modifier l'ensemble de transformations de haut-niveau. Comme ODT permet pas l'expression des relations entre les éléments du même espace multidimensionnel, ce choix aurait impliqué des modifications considérables dans tous les niveaux du formalisme ODT.

Nous avons choisi de prendre une autre direction, en se basant sur quelques observations sur les effets des transformations de haut-niveau sur la structure répétitive. Cela permet une séparation entre les transformations et les dépendances uniformes et nous ont amené à proposer un algorithme pour gérer l'interaction entre les deux concepts. Nous allons présenter en détail cette interaction dans le chapitre suivant.

6 IMPACT DU REFACTORING SUR LES DÉPENDANCES UNIFORMES

Nous avons présenté dans le chapitre précédent les transformations de haut niveau conçus autour le formalisme Array-OL qui permettent d'adapter à un haut niveau d'abstraction la spécification à l'exécution.

Dans ce chapitre nous analysons l'interaction entre les transformations Array-OL et l'extension que nous avons proposée pour l'expression des dépendances uniformes. L'intérêt de cette analyse est d'autoriser l'utilisation des techniques de refactoring sur des spécification qui contient des dépendances interrépétitions.

La section 6.1 analyse formellement cette interaction, propose et prouve un algorithme permettant de calculer les dépendances uniformes après une transformation Array-OL sans toucher au formalisme ODT. Un exemple pratique montrant le fonctionnement de l'algorithme est présenté par la suite dans la section 6.2 et nous finissons le chapitre par des conclusions sur l'interaction transformations/dépendances uniformes dans la section 6.3.

6.1 INTERACTION : ANALYSE FORMELLE

Les dépendances interrépétitions proposées dans la section 2.6, Modélisation des dépendances uniformes, ont été conçues dans le contexte d'Array-OL pour exprimer des dépendances uniformes entre des répétitions data-parallèles. Une telle construction exprime des dépendances de données entre toutes les répétitions qui se trouvent à une distance donnée par le vecteur de dépendance dans l'espace de répétition. Mais l'espace de répétition total pour une tâche est donné par la concaténation de toutes les répétitions hiérarchiques en partant du niveau le niveau le plus haut de la spécification vers le niveau de la tâche.

En connectant les dépendances à travers la hiérarchie, on peut exprimer des dépendances complexes entre des répétitions aux niveaux différents de la hiérarchie, très importants spécialement quand les transformations de haut niveau entrent en jeu. Le sujet à été déjà abordé dans sous-section 2.6.4 et sous-section 2.6.5 où nous avons vu comment une transformation de *tiling* qui partage l'espace de répétition en blocs nous a obligé à fissionner une dépendance initiale dans des dépendances in-

terconnectées entre différents niveaux de la hiérarchie et même d'avoir multiples dépendances sur le même niveau de la hiérarchie.

Dans la sous-section 2.6.5, nous avons vu comment en prenant la même spécification et en modifiant seulement le vecteur de dépendance initial qui exprimait des dépendances horizontales avec un pas de 2, $\mathbf{d} = (2, 0)$, vers un vecteur diagonal, $\mathbf{d} = (1, 1)$, après une transformation de tiling, les dépendances interrépétitions dans les deux cas sont significativement différentes, mais sans impact sur le reste du modèle (répétitions, motifs, tilers, etc.). Cette observation suggère une nette séparation entre les transformations Array-OL et les dépendances inter-répétition et nous a conduit à une analyse formelle de l'interaction qui a confirmé nos observations et nous a permis de proposer et prouver un algorithme qui gère cette interaction.

En prenant comme entrées les structures de l'application avant et après une transformation, ainsi que les dépendances initiales, l'algorithme est capable de calculer les dépendances exactes sur la nouvelle application qui respectent la contrainte de la fonctionnalité équivalente (la Définition 2, page 56).

Indépendamment de leur rôle, toutes les transformations de haut niveau Array-OL ont un impact similaire sur la structure de l'application, concernant les répétitions et la hiérarchie. En généralisant :

A. elles agissent sur la redistribution des répétitions à travers les niveaux hiérarchiques, avec la création ou la suppression des niveaux de la hiérarchie, si le cas ;

B. en plus, une transformation affecte au maximum deux niveaux successifs de répétitions hiérarchiques.

Définition 15 (niveaux des répétitions hiérarchiques)**.** Un niveau de répétition est représenté par la hiérarchie introduite par une tâche répétée. Plusieurs niveaux successifs de répétitions sont représentés par les décompositions successives en tâches répétées, ignorant les décompositions en tâches composées.

Observation. Deux niveaux successifs de répétitions peuvent être représentés soit par une décomposition tâche répétée–tâche répétée, soit tâche répétée–tâche composée–tâche répétée.

En prenant chaque transformation une par une, nous avons :

FUSION prend deux répétitions successives, crée un niveau supérieur de hiérarchie pour la répétition commune et les sous-répétitions qui restent sont placées sur le niveau inférieur de la hiérarchie, en minimisant la taille du tableau intermédiaire ;

TILING partage une répétition en blocs, en créant un niveau de hiérarchie ; le niveau supérieur pour la répétition des blocs et le niveau inférieur pour les répétitions du bloc ;

CHANGEMENT DU PAVAGE (soit par la création des dimensions, soit par agrandissement linéaire) agit en redistribuant des répétitions entre des niveaux successifs de la hiérarchie en en modifiant la granularité de l'application ;

APLATISSEMENT en étant l'opposé de la fusion et du tiling, supprime le niveau supérieur d'une hiérarchie et ajoute la répétition de ce niveau à toutes les répétions du niveau inférieur.

Pour résumer, une transformation prend comme entrées une structure d'un ou deux niveaux de répétitions et génère une autre structure d'un

ou deux niveaux de répétitions. En plus, une transformation garantit que les sémantiques de la spécification ne sont pas modifiés.

Après une transformation, les répétitions sont réarrangées dans les niveaux hiérarchiques et cela force un réarrangement des éventuelles dépendances inter-répétitions, pour qu'elles expriment les exactes mêmes dépendances entre répétitions qu'avant.

Le problème peut être formulé comme suit :

> *Ayant la structure avant et après la transformation (représentée les deux fois par une structure d'un ou deux niveaux successifs de répétitions), ainsi que la(les) dépendance(s) interrépétition avant la transformation, la(les) dépendance(s) interrépétition qui expriment les exactes mêmes dépendances sur les nouvelles répétitions transformées doivent être calculées.*

Dans cette démarche, des connexions entre les répétitions initiales et finales impliquées dans une transformation doivent être identifiées on manipulant le formalisme derrière ARRAY-OL et des contraintes qui assurent que les sémantiques de la spécification ne changent pas. Comme nous avons dit, une transformation effectue des modifications seulement dans les répétitions impliquées dans la transformation. L'interface avec le reste de l'application doit obligatoirement rester la même ; les tableaux qui communiquent avec le reste de l'application et les motifs de production/consommation doivent rester inchangés. Il est à travers ces tableaux qu'on peut identifier des connexions entre les répétitions avant et après une transformation.

Tout d'abord, nous allons commencer par la description des connexions entre les répétitions et les tableaux de l'interface dans les deux cas que nous intéressent : un ou deux niveaux hiérarchiques de répétitions.

6.1.1 *Un niveau de hiérarchie*

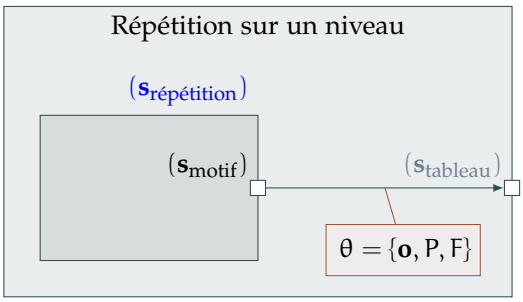


FIG. 65: Hiérarchie sur un niveau

Les connexions entre la répétition $\mathbf{s}_{\text{répétition}}$ et le tableau $\mathbf{s}_{\text{tableau}}$ sur la Figure 65 se fait à travers le tiler θ. En utilisant la définition de la construction pour les références des tuiles dans le tableau, *cf* l'Équation 2.2 sur la page 34, nous avons :

$$\forall \mathbf{r}, \mathbf{0} \leqslant \mathbf{r} < \mathbf{s}_{\text{répétition}}, \mathbf{réf}_{\mathbf{r}} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{tableau}} \tag{6.1}$$

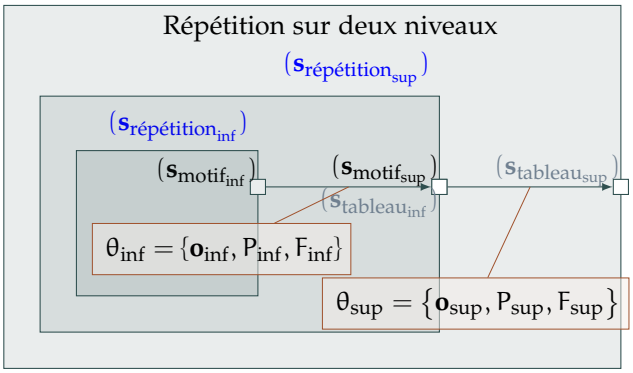


FIG. 66: Hiérarchie sur deux niveaux

6.1.2 *Deux niveaux de hiérarchie*

La connexion entre les deux espaces de répétition, $\mathbf{s}_{\text{répétition}_{\text{sup}}}$ et $\mathbf{s}_{\text{répétition}_{\text{inf}}}$, et le tableau $\mathbf{s}_{\text{tableau}}$ (Figure 66) se fait à travers les tilers θ_{sup} et θ_{inf}, et le tableau qu'ils partagent ($\mathbf{s}_{\text{motif}_{\text{sup}}} = \mathbf{s}_{\text{tableau}_{\text{inf}}}$).

$$\begin{aligned} &\forall \mathbf{r}_{\text{sup}},\ \mathbf{0} \leqslant \mathbf{r}_{\text{sup}} < \mathbf{s}_{\text{répétition}_{\text{sup}}}, \\ &\qquad \mathbf{réf}_{\mathbf{r}_{\text{sup}}} = \mathbf{o}_{\text{sup}} + P_{\text{sup}} \cdot \mathbf{r}_{\text{sup}} \bmod \mathbf{s}_{\text{tableau}_{\text{sup}}} \end{aligned} \tag{6.2}$$

$$\begin{aligned} &\forall \mathbf{r}_{\text{inf}},\ \mathbf{0} \leqslant \mathbf{r}_{\text{inf}} < \mathbf{s}_{\text{répétition}_{\text{inf}}}, \\ &\qquad \mathbf{réf}_{\mathbf{r}_{\text{inf}}} = \mathbf{o}_{\text{inf}} + P_{\text{inf}} \cdot \mathbf{r}_{\text{inf}} \bmod \mathbf{s}_{\text{tableau}_{\text{inf}}} \end{aligned} \tag{6.3}$$

Avec les deux tilers connectés à travers un tableau commun, en utilisant la construction de « court-circuit », qui permet l'expression directe des relations entre des éléments des tableaux connectés par plusieurs tilers successives, nous avons

$$\begin{aligned} &\forall \mathbf{r}_{\text{sup}},\ \mathbf{0} \leqslant \mathbf{r}_{\text{sup}} < \mathbf{s}_{\text{répétition}_{\text{sup}}}, \\ &\forall \mathbf{r}_{\text{inf}},\ \mathbf{0} \leqslant \mathbf{r}_{\text{inf}} < \mathbf{s}_{\text{répétition}_{\text{inf}}}, \\ &\qquad \mathbf{réf}_{\mathbf{r}_{\text{sup}}\mathbf{r}_{\text{inf}}} = \mathbf{o}_{\text{sup}} + F_{\text{sup}} \cdot \mathbf{o}_{\text{inf}} + P_{\text{sup}} \cdot \mathbf{r}_{\text{sup}} \\ &\qquad\qquad + F_{\text{sup}} \cdot P_{\text{inf}} \cdot \mathbf{r}_{\text{inf}} \mod \mathbf{s}_{\text{tableau}_{\text{sup}}} \end{aligned} . \tag{6.4}$$

En considérant les deux répétitions comme une seule, par la concaténation des deux espaces de répétition, la relation devient :

$$\begin{aligned} \forall \begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix}, \mathbf{0} \leqslant \begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix} < \begin{bmatrix} \mathbf{s}_{\text{répétition}_{\text{sup}}} \\ \mathbf{s}_{\text{répétition}_{\text{inf}}} \end{bmatrix}, \\ \mathbf{réf}_{\mathbf{r}_{\text{sup}}\mathbf{r}_{\text{inf}}} = \mathbf{o}_{\text{sup}} + F_{\text{sup}} \cdot \mathbf{o}_{\text{inf}} + (P_{\text{sup}}\ F_{\text{sup}} \cdot P_{\text{inf}}) \\ \cdot \begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix} \bmod \mathbf{s}_{\text{tableau}_{\text{sup}}} \end{aligned} \tag{6.5}$$

Dans les deux cas, *cf* l'Équation 6.1 et l'Équation 6.5, les relations de construction des motifs peuvent s'écrire sous la forme donne par l'Équation 6.1.

$$\forall \mathbf{r}, \mathbf{0} \leqslant \mathbf{r} < \mathbf{s}_{\text{rep}}, \mathbf{réf}_{\mathbf{r}} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{array}} \tag{6.6}$$

6.1.3 *Dépendances uniformes entre les répétitions*

La définition d'une dépendance inter-répétition (l'Équation 2.5, page 51) nous donne pour Équation 6.1 avec une dépendance uniforme, $\mathbf{d}$, la relation :

$$\begin{aligned}
&\forall \mathbf{r},\ \mathbf{0} \leqslant \mathbf{r} < \mathbf{s}_{\text{répétition}}, \mathbf{r}_{\text{dép}} = \mathbf{r} - \mathbf{d} \\
&\quad \mathbf{réf}_{\mathbf{r}_{\text{dép}}} = \mathbf{o} + \mathrm{P} \cdot \mathbf{r}_{\text{dép}} \bmod \mathbf{s}_{\text{tableau}} \\
&\Rightarrow \mathbf{réf}_{\mathbf{r}} - \mathbf{réf}_{\mathbf{r}_{\text{dép}}} = \mathrm{P} \cdot (\mathbf{r} - \mathbf{r}_{\text{dép}}) \\
&\Rightarrow \mathbf{d}_{\mathbf{réf}_{\mathbf{r}}} = \mathbf{réf}_{\mathbf{r}} - \mathbf{réf}_{\mathbf{r}_{\text{dép}}} = \mathrm{P} \cdot \mathbf{d}
\end{aligned} \tag{6.7}$$

D'après l'Équation 6.7, une dépendance uniforme entre les répétitions d'une tâche est équivalente avec une dépendance uniforme entre les références de ces répétitions dans un tableau ($\mathbf{d}_{\mathbf{réf}_{\mathbf{r}}}$).

6.1.4 *Analyse*

Nous avons montré comment dans les deux cas (un ou deux niveaux de hiérarchie), les relations entre les répétitions et leurs références dans un tableau peuvent être exprimées sous la même forme et une dépendance uniforme entre des répétitions est équivalente à une dépendance entre les références de ces répétitions dans un tableau.

De plus, la contrainte qui dit que la sémantique de l'application doit rester inchangée après une transformation laisse entendre que les tableaux à la frontière de l'action de la transformation doivent être produits de la même façon, et ainsi toutes les références des accès à un tel tableau avant la transformation doivent se retrouver après la transformation. *Une éventuelle dépendance uniforme entre les références des accès réguliers à un tableau doit rester inchangée.* C'est le lien qu'on cherchait pour connecter les dépendances sur les répétitions avant et après une transformation.

Maintenant, nous avons une structure initiale exprimée par une relation entre les répétitions et un tableau extérieur[1] ainsi que les dépendances initiales entre les références, introduites par la dépendance entre les répétitions

[1] À la frontière de l'action de la transformation.

$$\begin{aligned}
&\forall \mathbf{r}_{\text{avant}},\ \mathbf{0} \leqslant \mathbf{r}_{\text{avant}} < \mathbf{s}_{\text{répétition}_{\text{avant}}}, \\
&\qquad \mathbf{réf}_{\mathbf{r}_{\text{avant}}} = \mathbf{o}_{\text{avant}} + \mathrm{P}_{\text{avant}} \cdot \mathbf{r}_{\text{avant}} \bmod \mathbf{s}_{\text{tableau}}
\end{aligned} \tag{6.8}$$

$$\mathbf{d}_{\mathbf{réf}_{\text{avant}}} = \mathrm{P}_{\text{avant}} \cdot \mathbf{d}_{\text{avant}} \tag{6.9}$$

et une structure finale exprimée par une relation similaire,

$$\begin{aligned}
&\forall \mathbf{r}_{\text{après}},\ \mathbf{0} \leqslant \mathbf{r}_{\text{après}} < \mathbf{s}_{\text{répétition}_{\text{après}}}, \\
&\qquad \mathbf{réf}_{\mathbf{r}_{\text{après}}} = \mathbf{o}_{\text{après}} + \mathrm{P}_{\text{après}} \cdot \mathbf{r}_{\text{après}} \bmod \mathbf{s}_{\text{tableau}}
\end{aligned} \tag{6.10}$$

$$\mathbf{d}_{\mathbf{réf}_{\text{après}}} = \mathrm{P}_{\text{après}} \cdot \mathbf{d}_{\text{après}}, \tag{6.11}$$

en mettant la contrainte d'égalité entre les références avant et après, on se retrouve avec :

$$\mathbf{d}_{\mathbf{réf}_{\text{avant}}} = \mathbf{d}_{\mathbf{réf}_{\text{après}}} \Rightarrow \mathrm{P}_{\text{avant}} \cdot \mathbf{d}_{\text{avant}} = \mathrm{P}_{\text{après}} \cdot \mathbf{d}_{\text{après}}. \tag{6.12}$$

Résoudre l'Équation 6.12 suffit à trouver les dépendances sur les nouvelles répétitions :

A. s'il n'y a pas de solution, cela signifie qu'aucune dépendance uniforme n'existe qui exprime l'exacte même dépendance que celle avant la transformation et donc la sémantique de l'application ne peut pas être gardée, ce qui rendre la transformation incorrecte ;

B. une solution correspond a une dépendance sur le nouvel espace de répétitions ;

C. si plusieurs solutions existent, chaque solution correspond à une dépendance sur l'espace de répétition.

Si après la transformation la structure de l'application est représentée par un seul niveau de hiérarchie, chaque solution sera traduite dans une dépendance. Si par contre, la structure est représentée par deux niveaux de hiérarchie, chaque solution sera utilisée pour calculer les dépendances sur les deux espaces de répétition de chaque niveau de hiérarchie[2] :

[2] La séparation entre les deux dépendances se fait en conséquence avec les dimensions des espaces de répétition.

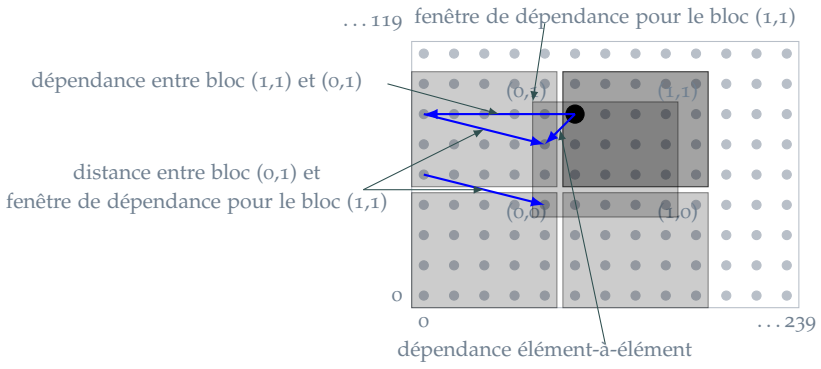
$$\mathbf{d}_{\text{after}} = \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ \mathbf{d}_{\text{inf}} \end{pmatrix} \tag{6.13}$$

Pour une solution :

A. si la dépendance du niveau supérieur, $\mathbf{d}_{\text{sup}}$, est le vecteur nul, pour cette solution il n'y a pas une dépendance sur le niveau haut et la dépendance au niveau inférieur sera représentée par le vecteur correspondant $\mathbf{d}_{\text{inf}}$;

B. si le vecteur $\mathbf{d}_{\text{sup}}$ n'est pas nul, nous avons des dépendances entre des éléments des blocs différents, représentées par une dépendance au niveau supérieur avec le vecteur de distance $\mathbf{d}_{\text{sup}}$ et les dépendances exactes élément à élément exprimées par un tiler au niveau inférieur de la hiérarchie, selon les règles de construction détaillée dans la sous-section 6.1.5.

La sémantique d'Array-OL pour les dépendances interrépétition force le passage des blocs entiers contenant des éléments en dépendance au niveau inférieur de la hiérarchie. Les dépendances exactes élément à élément seront représentées par un tiler similaire au tiler de sortie correspondant du niveau inférieur de la hiérarchie, avec une origine déplacée.



La dépendance élément-à-élément entre des répétitions placées dans des blocs différents se calcule comme somme des dépendance des deux niveaux de la hiérarchie.

FIG. 67: Dépendances élément-à-élément

6.1.5 *Calculer le déplacement de l'origine*

La dépendance élément à élément pour des répétitions des blocs différents sera exprimée comme somme des deux dépendances des deux niveaux de hiérarchie. Ayant les répétitions dépendantes en blocs différents, exprimer la dépendance au niveau inférieur avec une dépendance inter-répétition n'est plus nécessaire[3]. Utiliser une copie du tiler de sortie avec une origine déplacée suffit :

[3] et impossible à exprimer.

$$\forall\, \mathbf{r}_{\text{inf}}, \mathbf{0} \leqslant \mathbf{r}_{\text{inf}} < \mathbf{s}_{\text{répétition}_{\text{inf}}}, \mathbf{réf}_{\mathbf{r}_{\text{défaut}}} = \mathbf{o}_{\text{défaut}} + \mathrm{P}_{\text{inf}} \cdot \mathbf{r}_{\text{inf}} \tag{6.14}$$

La formule pour calculer le déplacement de l'origine peut être obtenue en imposant la contrainte d'identité des dépendances avant et après la transformation, même pour des répétitions des blocs différents, avec la dépendance exprimée par une relation de la forme donnée par l'Équation 6.14.

Pour une répétition $\begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix}$ qui dépend d'une autre répétition en dehors du son bloc, nous avons la référence dans le tableau d'origine

$$\mathbf{réf}_{\mathbf{r}_{\text{sup}}\mathbf{r}_{\text{inf}}} = \mathbf{o}_{\text{sup}} + \mathrm{F}_{\text{sup}} \cdot \mathbf{o}_{\text{inf}} + (\mathrm{P}_{\text{sup}}\ \mathrm{F}_{\text{sup}} \cdot \mathrm{P}_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix} \bmod \mathbf{s}_{\text{tableau}_{\text{sup}}} \tag{6.15}$$

et la référence de la répétition dépendante

$$\mathbf{réf}_{\mathbf{r}_{\text{dép}}} = \mathbf{o}_{\text{sup}} + \mathrm{F}_{\text{sup}} \cdot \mathbf{o}_{\text{def}} + (\mathrm{P}_{\text{sup}}\ \mathrm{F}_{\text{sup}} \cdot \mathrm{P}_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{r}_{\text{sup}} - \mathbf{d}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix} \bmod \mathbf{s}_{\text{tableau}_{\text{sup}}} \tag{6.16}$$

et ainsi la distance entre les deux est de

$$\begin{aligned} \mathbf{d}_{\mathbf{réf}_{\mathbf{r}}} &= \mathbf{réf}_{\mathbf{r}_{\text{sup}}\mathbf{r}_{\text{inf}}} - \mathbf{réf}_{\mathbf{r}_{\text{dép}}} \\ &= \mathrm{F}_{\text{sup}} \cdot (\mathbf{o}_{\text{inf}} - \mathbf{o}_{\text{def}}) + (\mathrm{P}_{\text{sup}}\ \mathrm{F}_{\text{sup}} \cdot \mathrm{P}_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ 0 \end{pmatrix}. \end{aligned} \tag{6.17}$$

Utilisant l'Équation 6.11 dans le cas d'une hiérarchie sur deux niveaux, on se retrouve avec

$$\mathbf{d}_{\mathbf{réf}_{\text{après}}} = \mathrm{P}_{\text{après}} \cdot \mathbf{d}_{\text{après}} = (\mathrm{P}_{\text{sup}}\ \mathrm{F}_{\text{sup}} \cdot \mathrm{P}_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ \mathbf{d}_{\text{inf}} \end{pmatrix} \tag{6.18}$$

et donc

$$\begin{aligned} & \mathbf{d}_{\mathbf{réf}_{\mathbf{r}}} = \mathbf{d}_{\mathbf{réf}_{\text{après}}} \\ \Rightarrow & (\mathrm{P}_{\text{sup}}\ \mathrm{F}_{\text{sup}} \cdot \mathrm{P}_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ \mathbf{d}_{\text{inf}} \end{pmatrix} \\ & = \mathrm{F}_{\text{sup}} \cdot (\mathbf{o}_{\text{inf}} - \mathbf{o}_{\text{défaut}}) + (\mathrm{P}_{\text{sup}}\ \mathrm{F}_{\text{sup}} \cdot \mathrm{P}_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ 0 \end{pmatrix} \\ \Rightarrow & \mathrm{P}_{\text{sup}} \cdot \mathbf{d}_{\text{sup}} + \mathrm{F}_{\text{sup}} \cdot \mathrm{P}_{\text{inf}} \cdot \mathbf{d}_{\text{inf}} \\ & = \mathrm{F}_{\text{sup}} \cdot (\mathbf{o}_{\text{inf}} - \mathbf{o}_{\text{défaut}}) + \mathrm{P}_{\text{sup}} \cdot \mathbf{d}_{\text{sup}} \\ \Rightarrow & \mathrm{P}_{\text{inf}} \cdot \mathbf{d}_{\text{inf}} = \mathbf{o}_{\text{inf}} - \mathbf{o}_{\text{défaut}} \end{aligned} \tag{6.19}$$

En conséquence, le déplacement de l'origine sera calculé utilisant $\mathbf{d}_{\text{inf}}$, selon la relation :

$$\mathbf{o}_{\text{défaut}} = \mathbf{o}_{\text{inf}} - \mathrm{P}_{\text{inf}} \cdot \mathbf{d}_{\text{inf}}, \tag{6.20}$$

où $\mathbf{o}_{\text{défaut}}$ représente l'origine du tiler du lien par défaut, $\mathbf{o}_{\text{inf}}$ et P_{inf} le vecteur d'origine et la matrice de pavage du tiler de sortie correspondent au niveau inférieur de la hiérarchie.

6.2 EXEMPLE PRATIQUE

Par la suite nous allons appliquer les résultats de l'analyse sur l'exemple de la dépendance diagonale (Figure 37, page 60) et la transformation de tiling en blocs de 5×4.

Nous avons une répétition initiale avec un espace de répétition de $\mathbf{r}_{\text{avant}} = \left(\begin{smallmatrix}240\\120\end{smallmatrix}\right)$, et une dépendance initiale de $\mathbf{d}_{\text{avant}} = \left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$, pendant que l'espace de répétition après la transformation de tiling en $(48, 30)$ blocs de $(5, 4)$ est $\left(\begin{smallmatrix}\mathbf{r}_{\text{sup}}\\\mathbf{r}_{\text{inf}}\end{smallmatrix}\right) = \left(\begin{smallmatrix}48\\30\\5\\4\end{smallmatrix}\right)$, avec une dépendance inter-répétition inconnue de $\left(\begin{smallmatrix}\mathbf{d}_{\text{sup}}\\\mathbf{d}_{\text{inf}}\end{smallmatrix}\right)$. Ayant $\mathrm{P}_{\text{après}} = \left(\mathrm{P}_{\text{sup}}\ \mathrm{F}_{\text{sup}} \cdot \mathrm{P}_{\text{inf}}\right)$ (*cf* l'Équation 6.5) et en remplaçant dans l'Équation 6.12 en utilisant le tableau de sortie comme connexion, nous avons le système d'inéquations :

$$\begin{cases} \begin{pmatrix}1 & 0\\0 & 4\end{pmatrix} \cdot \begin{pmatrix}1\\1\end{pmatrix} = \begin{pmatrix}5 & 0 & 1 & 0\\0 & 16 & 0 & 4\end{pmatrix} \cdot \begin{pmatrix}\mathbf{d}_{\text{sup}}\\\mathbf{d}_{\text{inf}}\end{pmatrix} \\ \begin{pmatrix}-\mathbf{r}_{\text{sup}}\\-\mathbf{r}_{\text{inf}}\end{pmatrix} = \begin{pmatrix}-48\\-30\\-5\\-4\end{pmatrix} < \begin{pmatrix}\mathbf{d}_{\text{sup}}\\\mathbf{d}_{\text{inf}}\end{pmatrix} < \begin{pmatrix}\mathbf{r}_{\text{sup}}\\\mathbf{r}_{\text{inf}}\end{pmatrix} = \begin{pmatrix}48\\30\\5\\4\end{pmatrix} \\ \begin{pmatrix}\mathbf{d}_{\text{sup}}\\\mathbf{d}_{\text{inf}}\end{pmatrix} \in \mathbb{Z}^4 \end{cases} . \tag{6.21}$$

Les solutions du système d'inéquations sont :

$$\left(\begin{smallmatrix}\mathbf{d}_{\text{sup}}\\\mathbf{d}_{\text{inf}}\end{smallmatrix}\right) \in \left\{ \left(\begin{smallmatrix}0\\0\\1\\1\end{smallmatrix}\right), \left(\begin{smallmatrix}1\\0\\-4\\1\end{smallmatrix}\right), \left(\begin{smallmatrix}0\\1\\1\\-3\end{smallmatrix}\right), \left(\begin{smallmatrix}1\\1\\-4\\-3\end{smallmatrix}\right) \right\} . \tag{6.22}$$

Les quatre solutions représentent les quatre dépendances sur la Figure 37. La première a $\mathbf{d}_{\text{sup}} = \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$, et ainsi la dépendance est sur le deuxième niveau de la hiérarchie avec un vecteur de $\mathbf{d}_{\text{inf}} = \left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$. Les trois autres solutions représentent des dépendances entre les blocs au niveau supérieur, avec un vecteur de dépendance égal à $\mathbf{d}_{\text{sup}}$, pendant que $\mathbf{d}_{\text{inf}}$ est utilisé pour calculer l'origine déplacée du lien par défaut sur le niveau inférieur de la hiérarchie, *cf* l'Équation 6.20. Pour ces trois dernières solutions, nous avons :

$$\mathbf{o}_{\text{défaut}} \in \left\{ \begin{pmatrix}4\\-1\\0\end{pmatrix}, \begin{pmatrix}-1\\3\\0\end{pmatrix}, \begin{pmatrix}4\\3\\0\end{pmatrix} \right\} . \tag{6.23}$$

Les solutions obtenues correspondent aux tilers d'accès qu'on retrouve sur l'exemple avec des triples dépendances entre les blocs de la Figure 37, sur la page 60.

6.3 CONCLUSIONS

Dans ce chapitre, nous avons analysé l'interaction entre les transformations de haut-niveau conçus autour le modèle de spécification ARRAY-OL et les dépendances interrépétition.

Comme une transformation affecte maximum deux niveau successives de répétitions hiérarchiques, nous avons montré comment, ayant la structure de l'application avant et après une transformation, nous pouvons calculer les dépendances uniformes qui expriment, sur la nouvelle structure, les mêmes dépendances élément-à-élément qu'avant la transformation. Cela garanti que la sémantique de l'application ne change pas et rend possible l'utilisation du refactoring sur des spécification contenant des dépendances uniformes.

L'algorithme proposé pour gérer l'interaction fonctionne indépendamment des transformation ARRAY-OL et cela facilite l'implémentation pratique : l'adaptation des dépendances est une étape ultérieure à la transformation ODT et n'influence pas les calculs d'ODT.

Dans cette partie de la thèse nous nous sommes intéressés aux techniques d'optimisations au niveau haut de la spécification répétitive. Nous avons vu l'ensemble d'outils de refactoring dans une perspective orientée plus théorique. Les extensions proposées sont aussi le résultat des travaux de validation dans le contexte de la comodélisation en MARTE pour GASPARD2.

Dans les chapitre suivantes nous allons présenter les outils de refactoring dans une perspective pratique, commençant avec l'étude de stratégies d'optimisation sur une application réelle de traitement intensif de signal. Les travaux d'implémentation dans le contexte de GASPARD2 seront présentés brièvement après.

Troisième partie

VALIDATION EXPÉRIMENTALE

7

STRATÉGIE DE REFACTORING POUR L'EXPLORATION DE L'ESPACE DE CONCEPTION

Dans les chapitres précédents, nous nous sommes intéressés à des techniques d'optimisation de haut niveau dans une perspective théorique. Nous allons essayer de voir comment les outils de refactoring peuvent être utilisés dans la pratique.

Des outils de refactoring sont disponibles pour adapter une spécification à l'exécution, en réduisant les tailles des tableaux et changeant la granularité du parallélisme pour permettre une expression directe d'un modèle d'exécution. Ces outils sont représentés par des transformations de haut-niveau data-parallèles dont nous avons discuté dans le chapitre 5, REFACTORING EN UTILISANT DES TRANSFORMATIONS DE HAUT NIVEAU.

Dans ce chapitre nous allons voir comment les outils de refactoring peuvent être utilisés pour adapter une spécification à l'exécution sur une application réelle de traitement intensif de signal, par l'exploration de l'espace de conception. L'application que nous allons utiliser est une application de traitement radar qui est le candidat idéal pour une modélisation en MARTE et pour l'exécution sur une architecture multiprocesseur. Dans ce sens, nous proposons une stratégie pour le refactoring de haut-niveau qui permet l'exploration de l'espace de conception des modèles répétitives MARTE, visant l'adaptation de la spécification aux contraintes d'exécution.

Dans la section 7.1, Nous commençons par la description de l'application de traitement radar. La stratégie d'optimisation basée sur les transformation de refactoring est approfondie dans la section 7.2 et une analyse des résultats est faite dans la section 7.3. Des conclusions sont présentés dans la section 7.4.

7.1 APPLICATION RADAR STAP

L'application que nous allons utiliser pour la modélisation et l'optimisation par du refactoring est une application industrielle de traitement radar MTI (de l'anglais Moving Target Indication), dont l'objectif est de détecter à partir d'un aéronef les objets en mouvement sur le sol, et spécialement en mouvement lent entre toutes les autres surfaces immobiles

reflétant l'onde du radar (échos de sol). Cela se fait par la réception de l'écho de sol d'une séquence périodique d'impulsions radar (rafales). Le traitement radar permet l'estimation en même temps de la position de la cible, par le délai entre la transmission du signal (pulse) et la réception de son écho, et de sa vitesse par l'effet Doppler qui affecte les échos d'une séquence de pulses identiques envoyés périodiquement : la vitesse de la cible se traduit par une (faible) variation de sa distance du radar, qui est visible uniquement comme un déplacement de phase du signal radar (e. g. environ 10GHz). Dans cette approche, le traitement Doppler consiste en un banc de filtres (e. g. Transformation de Fourier rapide), chacun réglé vers un déplacement particulier entre des échos successifs.

Ce type de traitement Doppler est dans certaines situations suffisant pour séparer les objets réfléchissants sur la base de leurs vitesses. Quand l'onde est orientée vers le sol, la partie principale de l'énergie de l'écho est censée provenir des objets qui composent le sol (appelés entassements), pendant que les objets d'intérêt envoient des échos faibles, mais avec un déplacement de phase. Cependant, l'onde radar n'est pas parfaitement nette et elle a une épaisseur de quelques degrés, qui aboutit à donner à certains objets immobiles sur le sol à la frontière de l'onde une vitesse relative à l'aéronef (causé par la vitesse propre de l'aéronef) et ainsi créant des interférences non désirées sur les échos des cibles en mouvement : cela crée une ambiguïté entre les vitesses intrinsèques et azimuts des cibles.

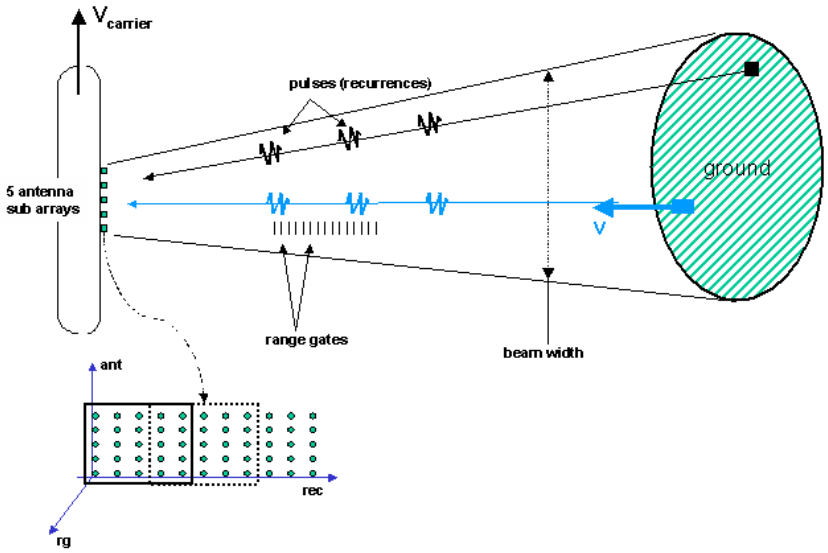


FIG. 68: STAP

Des techniques de filtrage adaptatif, où des filtres fixes sont remplacés par de filtres calculés au moment de l'exécution à partir du signal reçu lui-même, aident minimisant les influences de ce signal d'entassement indésirable : dans le cas de MTI, le traitement adaptatif espace-temps[1] est utilisé. Dans cette méthode, un ensemble de filtres est calculé à chaque rafale, par la résolution des systèmes linéaires avec la partie droite constituée des vecteurs de référence des motifs de phase théorétiques attendus sur les sous-tableaux de l'antenne pour plusieurs pulses consécutifs, chacun correspondant à une hypothèse particulière (vitesse, angle) de la cible relativement à l'aéronef. Cela est montré

[1] Space Time Adaptive Processing en anglais.

sur la Figure 68, où 2D motifs sur les dimensions de l'antenne et du pulse (rec) sont considérés pour calculer les filtres qui éliminent les ambiguïtés naturelles entre la vitesse et l'azimut.

7.1.1 *Contraintes d'implémentation*

Ces caractéristiques sont principalement :

1. une large partie de l'application est représentée sous la forme de flot de données, manipulant des tableaux multidimensionnels de données ;
2. la chaîne de traitement utilise différents opérateurs, chacun avec ses besoins particuliers en termes de précision et de plage dynamique ;
3. la chaîne de traitement est dynamique et peut varier pendant l'exécution par la variation fréquente des paramètres de l'algorithme (tailles des tableaux, bornes des boucles, ...), en préservant grosso modo le même graphe de traitement. Cela est généralement appelé multimodes dans la communauté radar ;
4. la charge en calcul est assez élevée pour clairement impliquer des unités matérielles de calcul parallèles ;
5. la performance temps réel est le besoin clé, à la fois en termes de débit de calcul et de latence. Cela peut résulter de certains besoins opérationnels et/ou des contraintes matérielles telles que des limitations en taille mémoire.

7.1.2 *Modélisation de* STAP

L'application STAP a été modélisée en Papyrus UML en utilisant le profile UML MARTE[2].

[2] Les figures de ce chapitre représentent des modèles UML utilisables.

Commençant au niveau le plus haut, l'application est successivement décrite en utilisant une décomposition composée ou répétitive, jusqu'au niveau de détails désiré, représenté par des tâches élémentaires qui peuvent être déployés sur des fonctions de bibliothèque.

Le niveau le plus haut de la spécification, illustré sur la Figure 69, décrit le fonctionnement global et l'interaction avec l'environnement.

Les dimensions infinies des tableaux traités par *GlobalSTAP* représentent l'abstraction du temps et la Figure 70 décrit le niveau du flot de données de l'application, la répétition infinie (le temps) d'un traitement STAP.

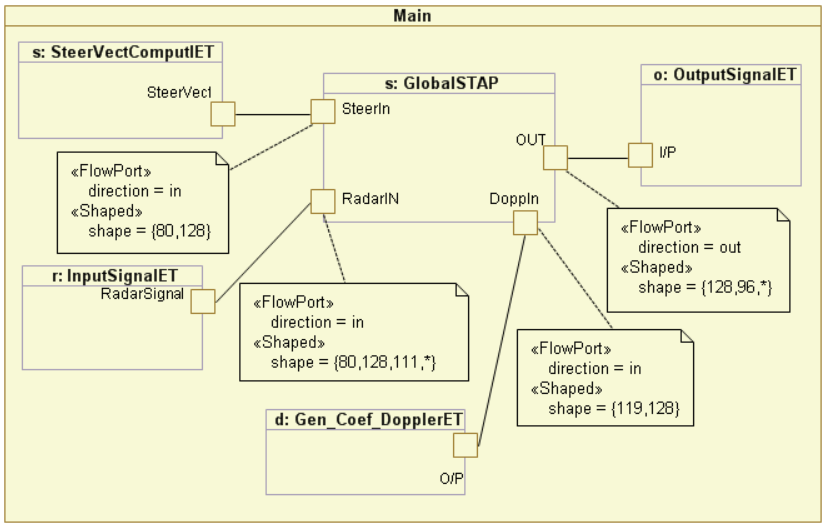
La Figure 71 illustre la décomposition de la tâche répétée en temps dans une succession de filtres répétitifs, avec les dimensions des tableaux et les espaces de répétition montrées sur la figure. Les tilers qui expriment les constructions des motifs/tuiles pour chaque répétition ne sont pas visibles[3].

[3] Un modèle complet est disponible à l'adresse `http://gforge.inria.fr/frs/download.php/5755/examples_papyrus.zip`.

Chaque filtre répétitif a une fonctionnalité élémentaire différente et accède distinctement à ses motifs :

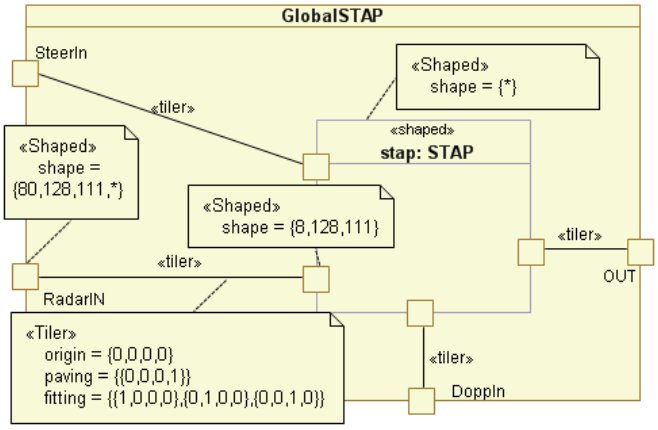
PULSECOMPRESSION prend des fenêtres glissantes monodimensionnelles de taille $\{16\}$ avec un pas de $\{1\}$ sur la troisième dimension du tableau d'entrée et calcule une valeur moyenne pour chaque de ces motifs ;

COVMATRIXESTIM prend $\{119, 96\}$ blocs de taille $\{10, 8\}$, en fenêtre glissante avec un pas de $\{1\}$ sur la deuxième dimension du tableau



La tâche principale, *GlobalSTAP*, prend comme entrées un tableau infini de données provenant des capteurs (*InputSignalET*) et, en utilisant les vecteurs de guidage fournis par *SteerVectCompIET* et les coefficients Doppler fournis par *Gen_Coef_DopplerET*, traite les données et fournit les résultats de la détection radar sous la forme d'un tableau infini, consommé par *OutputSignalET* pour l'affichage, la sauvegarde ou un traitement ultérieur.

FIG. 69: STAP : niveau haut



Seulement un tiler est montré, exprimant comment l'infinité des échos radar , représentés sous la forme d'un tableau avec une taille multidimensionnelle de $\{8, 128, 111, *\}$ est décomposé dans une infinité, $\{*\}$, de motifs avec une forme de $\{8, 128, 111\}$. La matrice de pavage de $\{\{0, 0, 0, 1\}\}$ désigne la correspondance entre la répétition infinie et la dernière dimension du tableau, pendant que la matrice d'ajustage $\{\{1, 0, 0, 0\}, \{0, 1, 0, 0\}, \{0, 0, 1, 0\}\}$ désigne les correspondances entre les dimensions du motif et les trois premières dimensions du tableau.

FIG. 70: STAP : le niveau flot de données

d'entrée et étend chaque motif vers un bloc de $\{80, 80\}$ qui sera rangé dans un tableau de taille $\{80, 80, 119, 96\}$;

AVERAGECOVAR élimine la troisième dimension du tableau d'entrée, en la remplaçant par la moyenne des $\{119\}$ valeurs de cette dimension ;

MATINV exécute l'inversion de $\{96\}$ matrices carrées de taille $\{80, 80\}$;

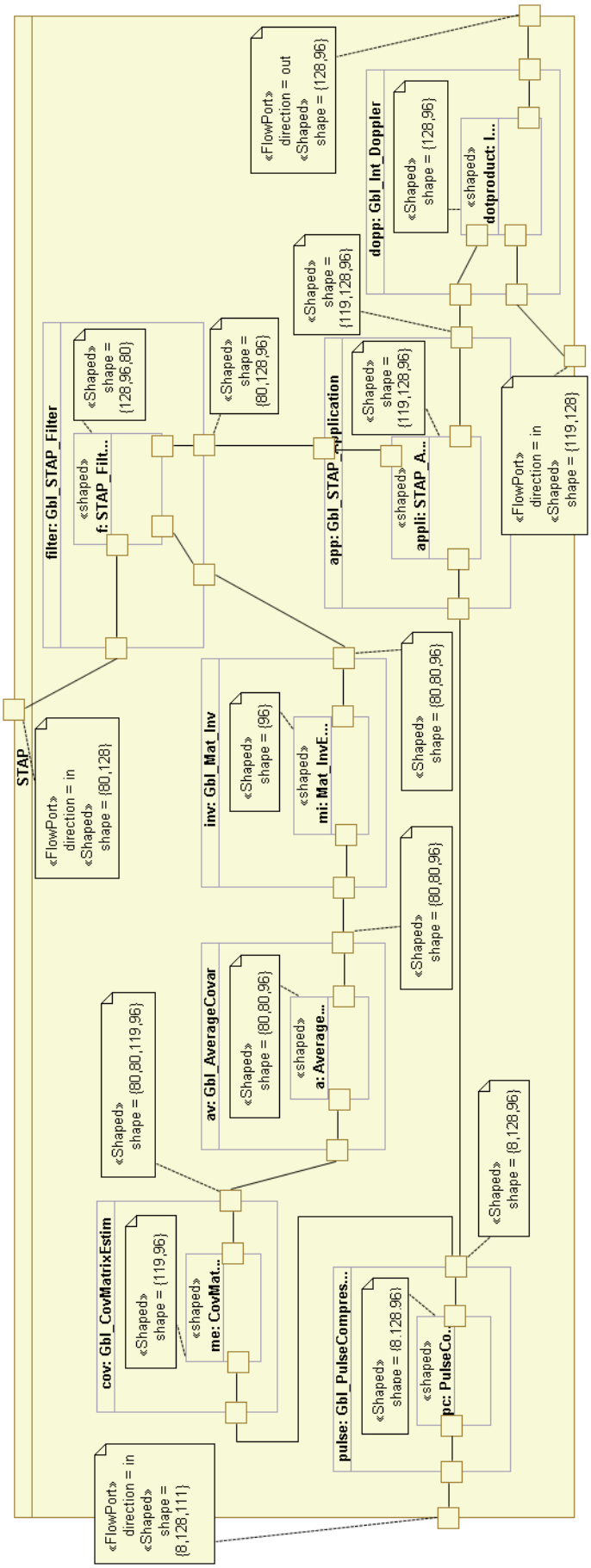


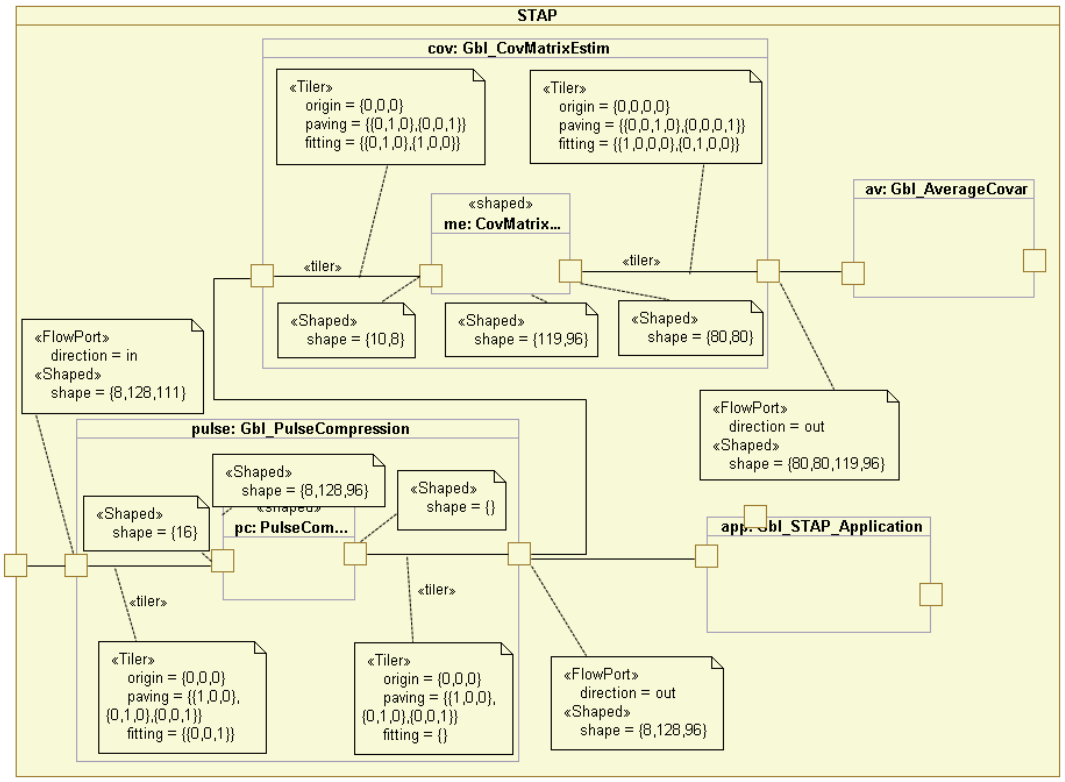
FIG. 71: Décomposition de STAP

STAP_FILTER applique {128} vecteurs de guidage, chacun avec une dimension de {80}, sur chaque ligne des matrices inversées, en produisant un tableau de taille {128, 96, 80} ;

STAP_APPLICATION compare les valeurs filtrées avec les pulses initiaux, par blocs de {8, 10}, pour identifier les mouvements ;

INT_DOPPLER applique les {128} vecteurs Doppler de taille {119} pour identifier les vitesses des cibles en mouvement.

Chaque répétition est caractérisée par des répétitions et des accès aux motifs uniformes différents, consommation ou production des données, exprimées par les informations de tiler sur chaque connecteur à l'intérieur d'une répétition, en concordance avec la relation spécifiée par l'Équation 2.3, sur la page 36. La Figure 72 illustre la construction des tilers pour deux des filtres STAP, *PulseCompression* et *CovMatrixEstim*.



- pour la répétition du *PulseCompression*, commençant avec chaque élément du tableau d'entrée (matrice de pavage identité), un motif de {16} éléments sur la troisième dimension est utilisé pour calculer une seule valeur, {}, valeurs réarrangées dans un tableau tridimensionnel avec une forme de {8, 128, 96} ;
- la différence de taille entre le tableau d'entrée et de sortie sur la troisième dimension ($111 - 96 = 15$) est due aux accès en fenêtre glissante, pour les 15 derniers éléments la fenêtre d'accès sort des dimensions du tableau et l'utilisation du modulo n'est pas souhaitée ;
- la répétition du *CovMatrixEstim* a aussi une fonctionnalité en fenêtre glissante, cette fois de blocs 2D avec une forme de {10, 8} glissant sur la deuxième dimension avec un pas de 1. Ces blocs sont étendus en {119, 96} blocs de taille {80, 80}.

FIG. 72: Tiler construction pour les deux premières filtres

Au niveau des filtres, chaque tâche répétée a un fonctionnement élémentaire et peut être déployée sur des fonctions de bibliothèque : inversion des matrices, des calculs des valeurs moyennes, etc.

Nos modèles sont statiques et ainsi nous avons choisi d'implémenter un seul mode de la fonctionnalité multimodes mentionnée en la descrip-

tion de l'application STAP de la sous-section 7.1.1. Des modes multiples peuvent êtres modélisés en utilisant les structures de contrôle basées sur les automates de mode proposées par Labbani *et al.* dans [55, 56] ou la prochaine version beta3 ou 1.0 de MARTE[4], mais sont en dehors l'objet de ce chapitre.

[4] Des références publique ne sont pas encore disponibles au moment de la rédaction.

7.2 VERS UN MODÈLE D'EXÉCUTION

Dans la section 4.1, Étude de l'exécution, nous avons discuté autour l'exécution d'une spécification répétitive. Pour rappeler, une *spécification* répétitive exprime le maximum de parallélisme, en utilisant une décomposition hiérarchique et répétitive. Elle décrit les dépendances de données entre les éléments des tableaux et, comme conséquence directe, un ordre partiel strict entre les appels des tâches. Un *modèle d'exécution* représente l'abstraction de l'exécution et il doit respecter l'ordre partiel strict défini par la spécification statique. Il est l'intention de la conception que, en exprimant l'ordre minimal d'exécution, de nombreuses décisions peuvent et doivent êtres prises au moment de l'association d'une spécification sur une plateforme d'exécution.

Nous avons choisi pour le passage d'une spécification MARTE RSM vers un modèle d'exécution de le faire le plus directement possible : *le modèle d'exécution devrait refléter la spécification*. Ce passage direct se traduit par la contrainte qui dit qu'une tâche peut commencer son exécution quand les tableaux d'entrées sont entièrement disponibles.

Des outils de refactoring basés sur des transformation de haut-niveau ont été conçus autour le formalisme pour permettre l'adaptation de la spécification aux contraintes d'exécution. Nous avons présenté ces transformation dans le chapitre 5. Ici, nous approchons le problème dans une perspective plus pratique et nous allons voir comment ces outils permettent l'exploration de l'espace de conception sur la modélisation de l'application STAP.

7.2.1 *Utilisation des transformations de haut-niveau data-parallèles*

Les transformations de code data-parallèles peuvent êtres utilisées dans l'adaptation de la spécification vers l'exécution, en permettant de choisir la granularité des flots comme une expression directe de l'association par l'étiquetage de chaque répétition avec son mode d'exécution : data-parallèle ou séquentiel.

Nos transformations de haut-niveau sont conçues comme outils de refactoring au niveau de la spécification pour optimiser l'application pour l'exécution sur des plateformes matérielles parallèles et pour l'exploration de l'espace de conception, qui mènent à quelques caractéristiques très intéressantes :

A. les transformations agissent comme des outils de refactoring, le résultat se traduit en changements du modèle au même niveau d'abstraction. Cela permet le chaînage des différentes transformations et facilite l'exploration de l'application, de l'architecture et du placement d'une sur l'autre ;

B. les transformations garantissent que l'application ne change pas de fonctionnalité, les mêmes valeurs de sortie seront calculées à partir des mêmes entrées ;

c. comme cibles pour l'optimisation, la réduction des tailles des tableaux et le management du parallélisme sont des priorités. Une spécification MARTE RSM exprime déjà le maximum de parallélisme et les transformations peuvent êtres utilisées pour changer la granularité de l'application pour une meilleure association sur une plateforme d'exécution parallèle ;

d. la spécification, le refactoring et le passage vers un modèle d'exécution sont des étapes séparées. La même spécification peut être traduite différemment vers l'exécution, en respectant les contraintes matérielles, fonctionnelles ou d'environnement.

7.2.2 *Refactoring utilisant des transformations de haut-niveau*

Par la suite, nous allons voir comment la spécification de STAP peut être adaptée à l'exécution en utilisant le refactoring et comment des stratégies de chaînage des transformations peuvent être déduites. Les concepts utilisés ont été discutés dans la présentation des transformations haut-niveau Array-OL (section 5.3) et les plus importantes seront rappelés.

ISOLATION DES DIMENSIONS INFINIES

La présence des dimensions infinies (dans les tailles des tableaux ou des répétitions) à travers l'application est la première préoccupation quand on fait le passage vers l'exécution. Une valeur infinie pour une répétition cause le blocage de l'exécution dans ce point et un tableau avec une dimension infinie ne peut pas être placé dans une mémoire avec une capacité bornée. Comme la dimension infinie est utilisée pour exprimer le temps (ou le flot de données), une solution est d'isoler les valeurs infinies à un niveau haut de la hiérarchie et de considérer ce niveau comme le niveau du flot de données à l'exécution : les tableaux ne seront pas placés entièrement en mémoire (seulement un motif à la fois) et une exécution séquentielle sera choisie pour la répétition.

En utilisant la transformation de *fusion* sur des répétitions successives infinies, une répétition commune peut être calculée et elle va représenter le niveau du flot de données à l'exécution[5].

[5] Une fusion des deux répétitions infinies qui échoue d'isoler la dimension infinie au niveau haut indique une spécification invalide.

L'application STAP a été déjà modélisée ayant toutes les valeurs infinies à un niveau de la hiérarchie (Figure 70) et donc cette étape peut être ignorée. La spécification de la même application, mais où les valeurs infinies auraient été au niveau de la décomposition en filtres aurait exprimé l'exacte même fonctionnalité, mais aurait nécessité l'isolation des valeurs infinies à un niveau flot de données avant le passage à l'exécution.

CHANGEMENT DE LA GRANULARITÉ

Des contraintes d'exécution pourraient imposer des changements dans la *granularité* de la spécification répétitive. Prenons, par exemple, si nous disposons de 4 processeurs et nous voulons exécuter la répétition sur la Figure 70 en parallèle, une transformation de *tiling* peut être utilisée pour partager l'espace de répétition en blocs de 4 répétitions qui peuvent êtres placées sur les 4 processeurs et exécutées en parallèle, pendant que la séquence de blocs du niveau haut représentera le niveau du flot de données. Cela représente ce que nous appelons changer la granularité d'une répétition et le résultat est illustré sur la Figure 73.

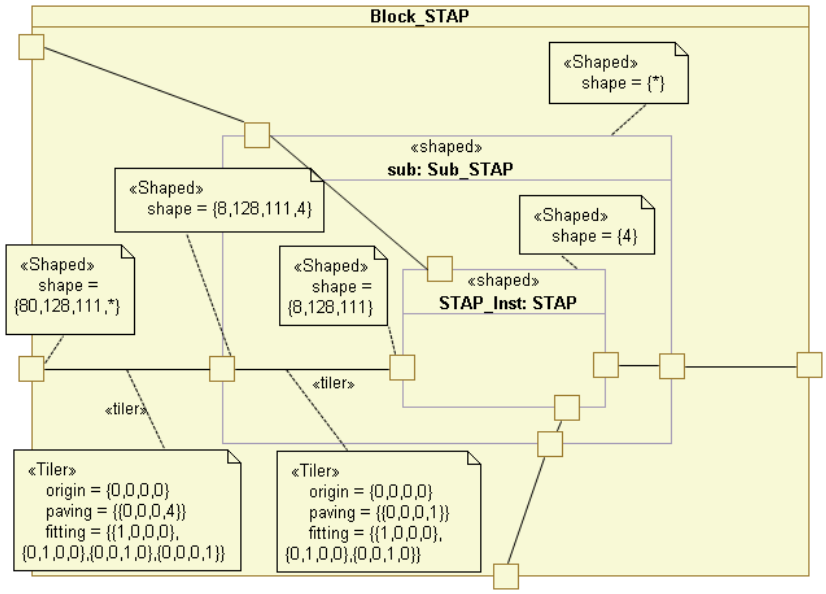


FIG. 73: Partage de la répétition infinie en blocs de $\{4\}$

RÉDUCTION DES TABLEAUX INTERMÉDIAIRES

La réduction des tableaux intermédiaires peut être réalisée par l'utilisation de la transformation de fusion et représente un des rôles principaux du refactoring. Une fusion élémentaire calcule un tableau intermédiaire minimal entre deux répétitions successives.

Définition 16 (Taille d'un tableau multidimensionnel)**.** La taille des tableaux est calculée en unités mémoires du type des éléments du tableau, par la multiplication des dimensions de sa forme multidimensionnelle.

Entre deux répétitions successives[6], un tableau intermédiaire minimal est désigné par un groupe minimal de motifs produit par la première tâche qui permet la deuxième répétition de s'exécuter au moins une fois, donc produire des éléments et en conséquence permettre une exécution sans blocage.

[6] La première produit un tableau consommé par la seconde.

Observation. La fusion calcule un groupe minimum de motifs intermédiaires et transforme l'application en créant une répétition commune avec deux sous-répétitions et un tableau intermédiaire réduit au groupe minimal de motifs.

Exemple. La Figure 74 montre le résultat de la fusion des deux répétitions de la Figure 72, *PulseCompression* et *CovMatrixEstim*. Une répétition commune avec une forme de $\{119, 96\}$ est calculée est deux sous-répétitions de $\{10, 8\}$ et respectivement $\{\}$ sur le deuxième niveau de la hiérarchie. Le groupe minimal de motifs produit par la première sous-répétition qui permet l'exécution au moins une fois[7] de la deuxième sous-répétition est de $\{10, 8, 1\}$, et ainsi le tableau intermédiaire est réduit de la taille initiale de $\{8, 128, 96\}$ (une demande en taille mémoire réduite de $8 \times 128 \times 96 = 98304$ éléments à $10 \times 8 \times 1 = 80$, avec un facteur de $98304/80 = 1228.8$ fois).

[7] Dans ce cas exactement une fois, $\{\}$.

La fusion de plusieurs répétitions connectées peut être réalisée par le chaînage de transformations élémentaires de fusion et d'aplatissement.

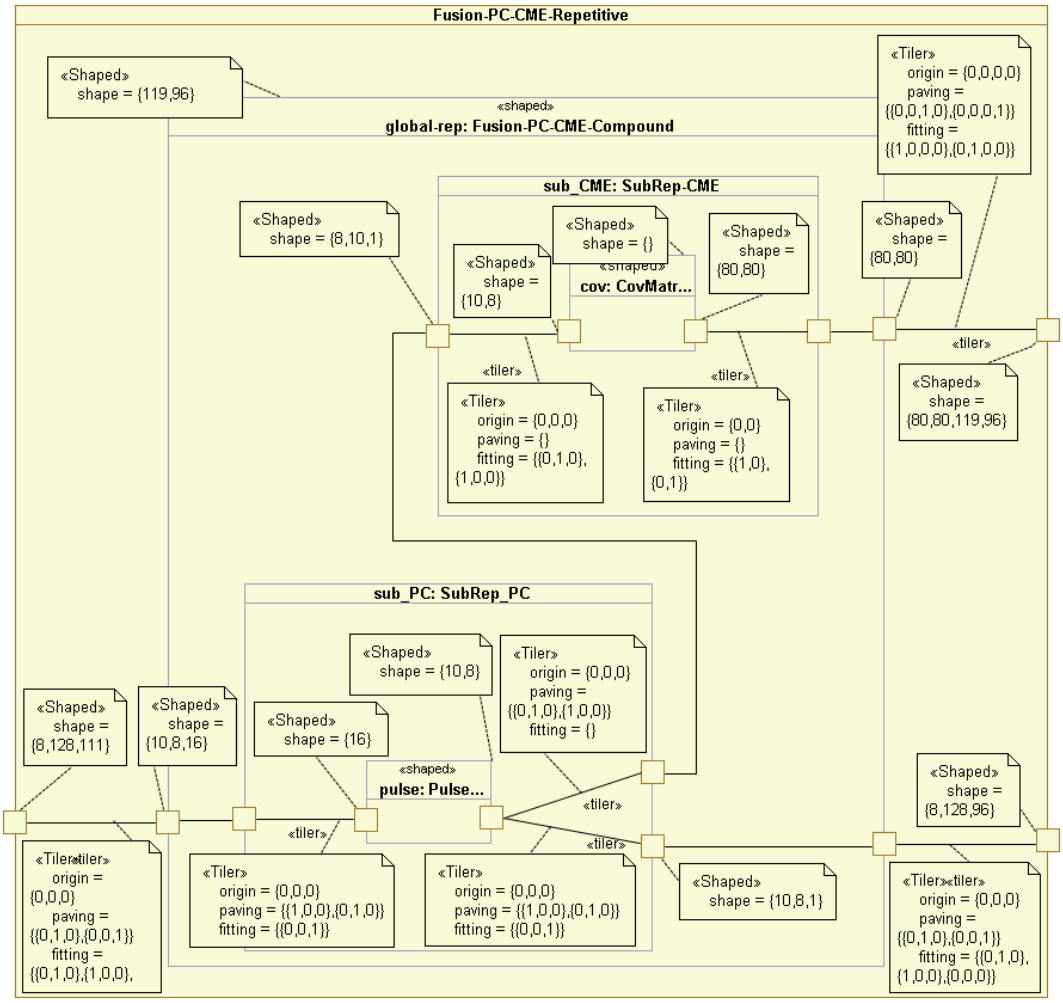


FIG. 74: Résultat de la fusion des deux premières répétitions

Suivant l'ordre de dépendance entre les tâches, chaque répétition est fusionnée avec le résultat de la précédente, pendant que la transformation d'aplatissement limite l'explosion des niveaux de hiérarchie.

[8] Selon l'ordre de la fusion.

Observation. Par une fusion multiple, seulement le dernier[8] tableau intermédiaire est minimisé. Les autres tableaux intermédiaires représentent des groupes minimaux de motifs nécessaires pour l'exécution au moins une fois de la dernière sous-répétition, et non de la sous répétition qui consomme le tableau respectif.

Exemple. La Figure 75 montre la fusion des répétitions *PulseCompression*, *CovMatrixEstim*, *AverageCovar* et *MatInv*. Les premières trois sous-répétitions représentent des exécutions minimales qui permettent la dernière sous-répétition de s'exécuter au moins une fois et ainsi minimiser le dernier tableau intermédiaire, mais pas les deux d'avant. Suite à une telle fusion multiple, le premier tableau intermédiaire est réduit à une taille de $\{10, 8, 1, 119\}$, 119 fois plus grande que le minimum tableau atteint par la fusion élémentaire des deux premières répétitions montrées sur la Figure 74.

Un autre facteur qui rentre en jeu est la quantité de *recalculs* introduits dans la spécification par la fusion. Les recalculs sont représentés par l'augmentation de la répétition complète pour la première tâche

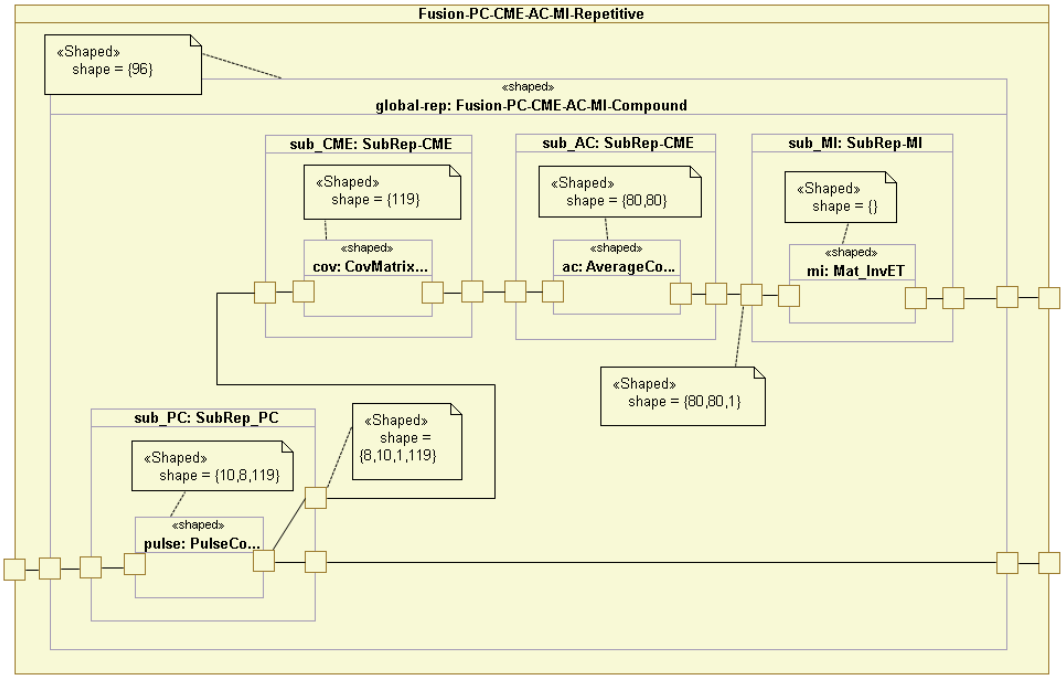


FIG. 75: STAP : fusion des quatre premières tâches

impliquée dans une fusion, causée par les motifs de production/consomption entre les deux répétitions.

Observation. La répétition complète d'une tâche est donnée par la concaténation de toutes les formes multidimensionnelles en descendant les niveaux de la hiérarchie du haut jusqu'au niveau de la tâche concernée[9]. Les recalculs sont représentés par l'augmentation de la répétition complète pour une tâche, suite à une transformation de fusion.

[9] La répétition infinie du niveau flot de données est négligée dans ce chapitre.

Définition 17 (Calcul associé à un espace de répétition)**.** Un espace de répétition multidimensionnel exprime une boucle d'exécution de la tâche répétée égale à la multiplication des dimensions de la forme multidimensionnelle de son espace.

Les accès en fenêtre glissante par la deuxième tâche impliquent dans une fusion, où des éléments initiaux se retrouvent dans plusieurs groupes de motifs minimaux après la fusion, détermine la première tâche de les calculer plusieurs fois, ce qu'explique le recalcul. Dans le cas des recalculs, la réduction de la taille du tableau intermédiaire a comme effet secondaire une augmentation des calculs.

Exemple. Sur la Figure 74, la répétition complète de la première tâche est, suite à la concaténation des espaces de répétition des deux niveaux de la hiérarchie, de $\{119, 96, 10, 8\}$, pendant que la répétition initiale était de $\{8, 128, 96\}$ et nous pouvons observer une augmentation des répétions avec un facteur de $119 \times 96 \times 10 \times 8/(8 \times 128 \times 96) = 9.29$.

Ainsi, dans le cas de valeurs intermédiaires qui sont utilisés plusieurs fois, le concepteur doit faire un compromis entre le stockage en mémoire ou des calculs multiples de ces valeurs.

7.2.3 *Réduction des tableaux intermédiaires de* STAP

La réduction des tableaux intermédiaires est l'utilisation principale de la transformation de fusion. Ayant plusieurs répétitions successives, nous cherchons une stratégie pour réduire au maximum l'ensemble des tableaux intermédiaires et en même temps d'éviter des points de blocage dans l'exécution et limitant le plus possible les recalculs introduits par les fusions. Des placements sur différentes architectures pourraient imposer légèrement différentes stratégies de refactoring, comme l'interdiction de recalculs ou la priorité vers la réduction des tailles des tableaux au détriment des recalculs. Sur l'application STAP de la Figure 71, nous avons choisi comme objectif de réduire les tableaux intermédiaires tout en limitant les recalculs.

FUSION COMPLÈTE. Une fusion *complète* (la fusion de toutes les répétitions successives d'un niveau de hiérarchie) n'est pas appropriée dans le cas de STAP, comme le montre le Tableau 3, contenant les valeurs des répétitions avant, après et les recalculs introduits par une telle fusion. Comme le montre le tableau, nous nous retrouvons avec une importante quantité de recalculs. La fusion multiple des répétitions garanti la minimisation seulement du dernier tableau intermédiaire. Le Tableau 2 montre comment seulement deux des tableaux intermédiaires ont une taille réduite, quant aux autres tableaux, nous pouvons observer même une augmentation en taille, causée par le re-arrangement des motifs suite à la succession des transformations fusion/aplatissement[10].

[10] Des accès initiaux chevauchants sont étendus sur plusieurs dimensions, avec duplication des éléments.

TABLEAU	TAILLE AVANT	TAILLE APRÈS	RÉDUCTION (/)
PC→CME	$8 \times 128 \times 96$	$8 \times 128 \times 119$	0,807
CME→AC	$80 \times 80 \times 119 \times 96$	$80 \times 80 \times 119 \times 119$	0,807
AC→MI	$80 \times 80 \times 96$	$80 \times 80 \times 119$	0,807
MI→SF	$80 \times 80 \times 96$	$80 \times 80 \times 119$	0,807
SF→SA	$80 \times 128 \times 96$	80×119	103,26
PC→SA	$8 \times 128 \times 96$	$8 \times 128 \times 119$	0,807
SA→ID	$119 \times 128 \times 96$	119	12288

Les tableaux sont identifiés par les tâches qui produisent/consomment le tableau (*Producteur→Consommateur*) et la réduction de la taille est donnée par la division des tailles des tableaux avant et après le refactoring.

TAB. 2: Tailles des tableaux pour la fusion complète

Néanmoins, une telle transformation apporte des informations utiles qui peuvent être utilisées pour trouver la meilleure séquence de fusions pour atteindre notre objectif déclaré. Tout d'abord, elle fournit l'ordre de fusion qui représente l'ordre partiel strict entre les tâches répétées, dans ce cas : *PulseCompression* $\oplus$ *CovMatrixEstim* $\oplus$ *AverageCovar* $\oplus$ *MatInv* $\oplus$ *StapFilter* $\oplus$ *StapApplication* $\oplus$ *IntDoppler*. Deuxièmement, sur le Tableau 3 nous pouvons identifier où des changements dans les recalculs apparaissent et ainsi les fusions qui introduisent des recalculs, dans nos cas : *MatInv* $\oplus$ *StapFilter* et *StapFilter* $\oplus$ *StapApplication*. Dans notre but de limiter les recalculs, cela peut suggérer que la fusion des quatre premières répétitions et des deux dernières, tout en isolant *StapFilter*, n'introduirait pas des recalculs.

TÂCHE	RÉPÉTITION AVANT	RÉPÉTITION APRÈS	RECALCUL (×)
PULSECOMPRESSION (PC)	$8 \times 128 \times 96$	$128 \times 96 \times \left(\begin{array}{c} 8 \times 128 \times 119 \\ 119 \times 119 \\ 80 \times 80 \times 119 \\ 119 \\ 80 \times 119 \\ 119 \\ 1 \end{array} \right)$	119×128
COVMATRIXESTIM (CME)	119×96		119×128
AVERAGECOVAR (AC)	$80 \times 80 \times 96$		119×128
MATINV (MI)	96		119×128
STAPFILTER (SF)	$128 \times 96 \times 80$		119
STAPAPPLICATION (SA)	$119 \times 128 \times 96$		1
INTDOPPLER (ID)	128×96		1

La hiérarchie de répétitions est représentée par des parenthèses dans le tableau. Après la fusion complète des filtres, une répétition commune de $\{128, 96\}$ est calculée et la répétition totale pour chaque des sous-répétitions est donnée par la concaténation avec celle commune. Le recalcul est calculé comme division entre la quantité des répétitions après et avant le refactoring.

TAB. 3: Répétitions pour la fusion complète

FUSIONS DEUX-PAR-DEUX. D'autres fusions qui apportent des informations utiles sont les fusions de chaque deux répétitions successives. Le résultat contient les valeurs des tableaux minimaux réalisables entre deux répétitions et sur le recalcul introduit par une telle opération, comme montré sur le Tableau 4.

FUSION	RÉDUCTION (/)	RECALCUL (×)
PULSECOMPRESSION ⊕ COVMATRIXESTIM	$128 \times 96/10 = 1228.8$	9.29
COVMATRIXESTIM ⊕ AVERAGECOVAR	96	1
AVERAGECOVAR ⊕ MATINV	96	1
MATINV ⊕ STAPFILTER	96	128
STAPFILTER ⊕ STAPAPPLICATION	$128 \times 96 = 112288$	119
PULSECOMPRESSION ⊕ STAPAPPLICATION	$128 \times 96/10 = 1228.8$	$119 \times 10 = 1190$
STAPAPPLICATION ⊕ INTDOPPLER	$128 \times 96 = 12288$	1

La réduction maximale pour les tableaux intermédiaires ainsi que le recalcul introduit sur la première sous-répétition, après la fusion de chaque deux répétitions successives.

TAB. 4: Fusion deux-par-deux

MEILLEUR CHOIX DE FUSION. Les informations apportées par la fusion complète et deux-par-deux peuvent guider le choix pour une séquence de transformations qui achèvera les meilleurs résultats pour notre objectif choisi de réduire au maximum les tableaux intermédiaires tout en limitant les recalculs. La fusion complète suggère de grouper les quatre premières répétitions et les deux dernières, mais les fusions deux-par-deux montrent que la fusion des deux premières répétitions introduit des recalculs avec un facteur de presque 10 pour la réduction du tableau intermédiaire avec un facteur de 1228.8.

Nous avons deux choix pour le premier groupe de répétitions :

A. si le facteur de 10 pour le recalcul de la première sous-répétition est acceptable, nous pouvons grouper les quatre premières répétitions pour la fusion ;

B. sinon, seulement les répétitions deux à quatre seront fusionnées, sans recalcul cette fois.

Pour illustration, nous avons choisi la première alternative, de grouper les premières quatre répétitions et les deux dernières. Le résultat mon-

tre une réduction au maximum (selon les tailles minimales réalisables du Tableau 4) des tableaux : *CovMatEst→AvCov*, *AvCov→MatInv* and *StapApp→IntDopp* mais une réduction non-maximale pour *PulCompr→CovMatEst*. Sur le deuxième niveau de la hiérarchie, par la fusion des deux sous-répétitions, *PulseCompresion⊕CovMatrixEstim*, nous pouvons réduire encore plus ce tableau (au maximum cette fois), avec comme résultat la présence de trois niveaux de hiérarchie pour ces deux sous-répétitions. Les réductions en taille des tableaux sont montrés sur le Tableau 5, pendant que les répétitions et les recalculs sont montrés sur le Tableau 6.

TABLEAU	TAILLE AVANT	TAILLE APRÈS	RÉDUCTION (/)
PC→CME	$8 \times 128 \times 96$	8×10	1228.8
CME→AC	$80 \times 80 \times 119 \times 96$	$80 \times 80 \times 119$	96
AC→MI	$80 \times 80 \times 96$	80×80	96
MI→SF	$80 \times 80 \times 96$	$80 \times 80 \times 96$	1
SF→SA	$80 \times 128 \times 96$	$80 \times 128 \times 96$	1
PC→SA	$8 \times 128 \times 96$	$8 \times 128 \times 96$	1
SA→ID	$119 \times 128 \times 96$	119	12288

TAB. 5: Tailles des tableaux pour le meilleur choix de fusion

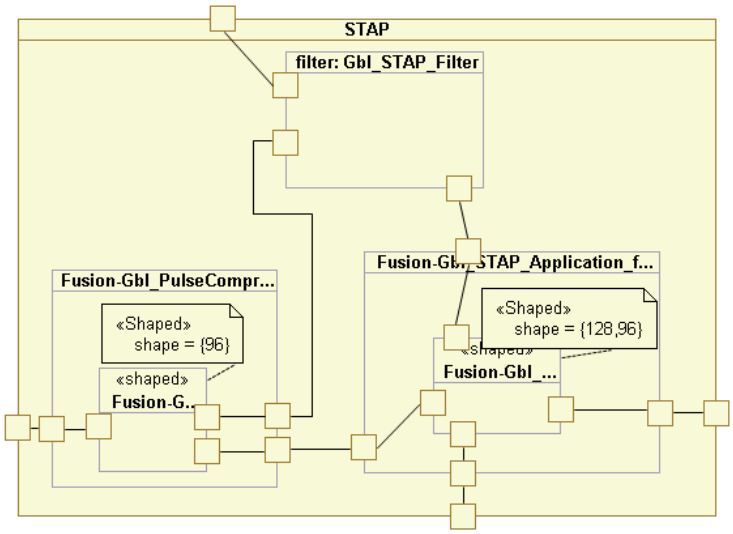
TÂCHE	RÉPÉTITION AVANT	RÉPÉTITION APRÈS	RECALCUL (×)
PULSECOMPRESSION	$8 \times 128 \times 96$	$96 \times \begin{pmatrix} 119 \times \begin{pmatrix} 10 \times 8 \\ 1 \end{pmatrix} \\ 80 \times 80 \\ 1 \end{pmatrix}$	9.29
COVMATRIXESTIM	119×96		1
AVERAGECOVAR	$80 \times 80 \times 96$		1
MATINV	96		1
STAPFILTER	$128 \times 96 \times 80$	$128 \times 96 \times 80$	1
STAPAPPLICATION	$119 \times 128 \times 96$	$128 \times 96 \times \begin{pmatrix} 119 \\ 1 \end{pmatrix}$	1
INTDOPPLER	128×96		1

TAB. 6: Répétitions pour le meilleur choix de fusion

Une partie de l'application transformée selon la stratégie précédente est illustré sur la Figure 76, le niveau des filtres et, sur la Figure 77, les deux dernières sous-répétitions.

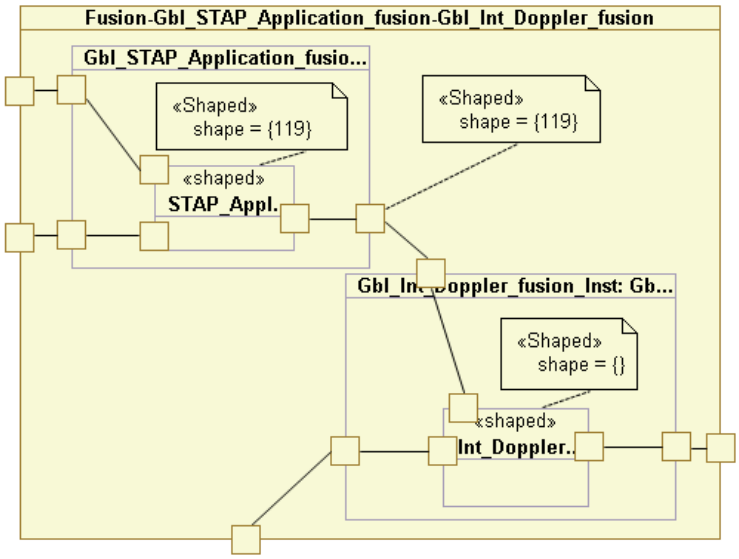
7.3 ANALYSE DES RÉSULTATS

Dans la section précédente, nous avons vu comment les transformations de haut niveau peuvent être utilisées pour adapter une spécification à l'exécution. Les optimisations sont de deux types, plateforme dépendante ou d'usage général. La réduction des tableaux intermédiaires peut être vue comme une optimisation d'usage général, mais comme montré, des contraintes peuvent guider le choix des fusions qui dirigent la réduction des tableaux intermédiaires. Des changements de la granularité, en utilisant des transformations de changement de pavage, tiling ou aplatissement, sont des transformations plus orientées optimisations plateforme-dépendante, visant l'adaptation de la spécification vers l'architecture matérielle et l'optimisation du placement des répétitions sur les unités parallèles.



Les premières quatre répétitions groupées dans une seule répétition commune de $\{96\}$ et les deux dernières dans une répétition de $\{128, 96\}$ d'une tâche composée contenant les deux sous-répétitions des tâches initiales, montrées sur la Figure 77.

FIG. 76: Niveau des filtres transformé



La réduction du tableau intermédiaire *StapApp→IntDopp* vers une taille de 119 peut être observée.

FIG. 77: Sous-répétitions de *StapApplication* et *IntDoppler*

Les applications de traitement intensif de signal sont souvent représentées sous la forme de filtres successifs, comme le cas de l'application radar STAP. Les accès complexes aux motifs rendent impossible la tâche de réduire au maximum tous les tableaux intermédiaires et parfois seulement avec l'introduction des calculs supplémentaires.

Nous avons montré comment, en explorant des scénarios variés de refactoring, nous pouvons extraire des informations qui nous amènent vers le meilleur choix d'enchaînement de transformations. La fusion

complète apporte des informations sur l'ordre partiel strict et sur les jonctions où des recalculs sont introduits, pendant que les fusions deux-par-deux fournissent les valeurs pour les réductions maximales réalisables et sur les recalculs introduits par une telle réduction. Ces informations ont été utilisées pour séparer les répétitions en groupes en interdisant des fusions qui introduisent beaucoup de recalculs. Des objectifs et des applications différentes pourront déterminer l'usage alternatif des fusions, prenant par exemple la présence d'un tableau à dimension infinie, l'élimination de cette dimension infinie est prioritaire à l'introduction des recalculs.

7.4 CONCLUSIONS

Nous avons montré dans ce chapitre comment les transformations de haut-niveau data-parallèles peuvent être utilisées dans l'exploration de l'espace de conception pour des applications multidimensionnelles de traitement intensif de signal, modélisées en MARTE RSM. Le but est de restructurer la modélisation à un niveau haut de spécification pour permettre un passage direct à l'exécution. La direction principale d'utilisation est la réduction des tailles des tableaux et le changement dans la granularité des répétitions en fonction des contraintes matérielles, du placement, de l'environnement, de l'exécution, etc.

Nous avons présenté une stratégie qui permet la réduction des tailles des tableaux intermédiaires guidée par les accès aux motifs. Des autres stratégies peuvent être utilisées et, ensemble avec des placements variés sur la plateforme d'exécution, elles peuvent être explorées et guidées avec des résultats remontés des niveaux de la génération de code, de la simulation ou de l'exécution dans l'environnement GASPARD2.

Comme travaux d'avenir, ces différentes stratégies pourront être implémentées dans l'environnement, ainsi que des outils semi-automatiques de refactoring : proposition pour le concepteur de différentes stratégies avec des gains calculés en matière de réduction des tableaux et de recalculs.

Dans le chapitre suivant, nous allons présenter brièvement les travaux d'implémentation et intégration dans l'environnement GASPARD2.

8 REFACTORING DANS LE CONTEXTE DE GASPARD2

Nous avons vu dans les chapitres précédents la base théorique derrière le refactoring de haut-niveau basé sur des transformations data-parallèles et comment ces outils peuvent être utilisés pour l'exploration de l'espace de conception pour adapter une spécification aux contraintes d'exécution. Des stratégies de refactoring peuvent être explores, comme celle présentée dans le chapitre précédent, STRATÉGIE DE REFACTORING POUR L'EXPLORATION DE L'ESPACE DE CONCEPTION, pour la réduction des tailles des tableaux intermédiaires.

Pour pouvoir bénéficier des outils de refactoring et aussi pour pouvoir les utiliser en collaboration avec les autres outils autour l'environnement de comodélisation GASPARD2, ils doivent être intégrés dans l'environnement.

Pendant cette thèse, une large partie des travaux a visé l'intégration des outils de refactoring dans GASPARD2 et l'interfaçage avec l'ensemble des outils développés au sein de l'équipe. L'intégration permet en même temps la validation des travaux théoriques, mais aussi nous a permis d'avoir un retour et d'étendre les outils de transformations dans une vision orientée plus vers l'utilisation pratique.

Dans ce chapitre nous allons présenter brièvement les travaux d'implémentation réalisés pendant la thèse. Il s'agit principalement de l'implémentation des transformations de haut-niveau et de l'interfaçage avec l'environnement, mais aussi des travaux dans le contexte IDM de GASPARD2, notamment autour les transformations de modèles entre les différents niveaux d'abstraction de GASPARD2.

La section 8.1 résume mes contributions relativement à l'implémentation et l'intégration dans l'environnement GASPARD2. La section 8.2 identifie quelques perspectives d'évolution et la section 8.3 présente les conclusions.

8.1 CONTRIBUTIONS

Les optimisations de haut-niveau data-parallèles ont été conçues dans le contexte du modèle de calcul ARRAY-OL et sont utilisées dans l'environnement de comodélisation GASPARD2. La technique d'optimisation s'appuie principalement sur l'exploration de l'espace de conception en suivant plusieurs directions, selon l'objectif, les contraintes et les caractéristiques de la spécification.

Dumont a commencé les travaux d'implémentation des transformations ARRAY-OL par l'implémentation d'un cadre permettant la manipulation du formalisme ODT dans le contexte des transformations ARRAY-OL. L'interfaçage avec le niveau des opérations ODT se faisait par l'intermède d'une *interface de programmation* (API[1]) qui permet la spécification des structures de répétitions hiérarchiques interconnectées, la transformation de ces structures dans un format qui corresponde au formalisme ODT et l'appel à la transformation ODT. Le résultat de la transformation ODT est traduit sous la forme de la spécification interne de l'API.

[1] De l'anglais Application Programming Interface.

Malgré cette interface avec les transformations ODT, il n'y avait pas un vrai branchement dans l'environnement GASPARD2 et l'utilisation des transformations était assez laborieuse.

Pendant ma thèse, une large partie de mes activités ont été orientées vers le branchement de ces transformations de haut-niveau au GASPARD2. Cela a impliqué :

- extension des transformations ODT : seule la transformation de fusion était implémentée auparavant ;
- implémentation des extensions proposées dans la section 5.4 ;
- branchement au niveau du métamodèle GASPARD2 ;
- avec le passage au profil UML standard de MARTE, le branchement sur l'éditeur visuel Papyrus UML ;
- implémentation de l'interaction avec les dépendances interrépétitions ;
- validation des travaux sur de modélisations d'applications réelles ;
- dans le cadre de la méthodologie IDM de GASPARD2, participation aux différentes étapes du développement : définition des métamodèles intermédiaires, implémentation de transformations de modèles, . . . ;
- dans le cadre du projet Ter@ops, l'interfaçage avec le reste de l'atelier par l'import/export entre nos modèles (UML + profile MARTE) et le format XML proposé par PIPS[2].

[2] http://www.cri.ensmp.fr/pips/

8.1.1 *Niveau transformations*

Le passage entre une spécification de haut-niveau en GASPARD2 et le formalisme ODT pour les opérations nécessaires aux calculs des transformations se fait à travers trois niveaux :

NIVEAU SPÉCIFICATION : la spécification basée sur la décomposition en composants répétitifs hiérarchiques ;

NIVEAU API : à ce niveau les répétitions et les dépendances de données sont identifiées et la traduction vers le formalisme ODT et à l'envers est gérée ;

NIVEAU ODT : les répétitions sont exprimées sous la forme ODT, des matrices et des vecteurs de valeurs rationnels ensemble avec tous les opérateurs ODT et les opérations du formalisme.

Au niveau de l'API les répétitions sont représentées comme des objets *OdtApiTask* et chaque tiler est transformé dans une *OdtApiDependance*, d'entrée ou de sortie, selon le cas. Une telle dépendance contient toutes les informations de l'accès par motifs uniformes (origine, matrices de pavage et d'ajustage). Une *OdtApiTask* peut être hiérarchique, cas ou les sous-répétitions sont référencées sous la forme d'une liste d'*OdtApiTask*s. Les dépendances de données entre les répétitions s'-

expriment par des points de connexion, *OdtApiConnectionPoint* : deux dépendances qui consomment/produisent le même tableau partagent le même *OdtApiConnectionPoint*.

Chaque transformation a sa propre interface d'appel. *OdtApiFusion* gère la transformation de fusion entre deux répétitions connectées : deux *OdtApiTask* qui partagent un point de connexion entre deux dépendances, de sortie pour la première et d'entrée pour la deuxième.

À ce niveau j'ai ajouté les transformations ODT qui n'étaient pas encore implémentées :

- *OdtApiChangementPavage* pour les transformations de changement de pavage par ajout de dimensions et par agrandissement linéaire ;
- *OdtApiCollapse* pour l'aplatissement ;
- *OdtApiTiling* pour l'opération de tiling.

8.1.2 *Niveau interface*

Pour pouvoir utiliser les transformations Array-OL dans le contexte de la spécification en Gaspard2, le branchement de la boîte à outils de refactoring à l'environnement de comodélisation était impératif. Avec l'aide des outils de développement autour Eclipse, une interface utilisateur branchée à l'éditeur de modèles a été développée.

Au début, le branchement se faisait au niveau des modèles `ecore` conformes au métamodèle Gaspard2, la première étape disponible sous Eclipse au niveau spécification[3].

[3] La modélisation visuelle se faisait en MagicDraw UML.

Avec l'apparition du profile UML standard OMG MARTE, des efforts ont été faits dans l'équipe pour s'aligner au standard par l'adoption du standard comme norme de spécification. Dans cette direction nous avons profité aussi de l'éditeur Papyrus UML, basé aussi sur la plateforme Eclipse. Maintenant nous pouvons vraiment dire que nous disposons d'un environnement intégré de comodélisation : toutes les étapes, de la spécification visuelle jusqu'à la génération du code en passant par les différents niveaux d'abstraction, sont intégrées autour la plateforme Eclipse.

Pouvoir profiter de l'interface visuelle disponible dans Papyrus UML pour les transformations de haut niveau nous semblait indispensable. Une autre interface avec la boîte de refactoring a été développée, cette fois branchée au éditeur visuel de Papyrus. Les principes de l'interface restent les mêmes pour les deux interfaces :

- en fonction d'/des élément(s) sélectionné(s), des transformations de haut niveau sont mises à disposition ;
- l'utilisateur choisit la transformation voulue, en spécifiant des paramètres si le cas : sur quelle(s) dimension(s) de faire le changement de pavage, le facteur de pavage, etc. ;
- la partie de la spécification qui est impliquée dans la transformation est transformée sous la forme de l'API.
  Observation. Maximum deux niveaux de répétitions hiérarchiques commençant avec la/les tâche(s) sélectionnée(s) sont nécessaires ;
- l'API s'occupe du calcul de la transformation au niveau ODT et de mettre à disposition le résultat de la transformation ;
- le résultat, toujours sous la forme d'une hiérarchie de maximum deux niveaux de répétitions, est rétransformé dans des objets qui correspondent au modèle initial.

Observation. Dans le cas de l'interface avec l'éditeur visuel de Papyrus UML, les représentations visuelles des éléments transformés doivent être transformées aussi.

– les objets résultat sont branchés au bon endroit dans le modèle UML.

La traduction des éléments du modèle sélectionné est une opération assez simple, en faisant la traduction un-à-un de tous les éléments équivalents. L'équivalence des concepts à été discutée dans la sous-section 3.6.1, Équivalence des concepts.

Le problème plus complexe est de retrouver les bons endroits où réinsérer les éléments transformés. Cela se fait en gardant un lien entre les éléments avant et après la transformation, par des relations entre certains concepts de la forme interne de l'API et les concepts du modèle initial :

– les *OdtApiDependence*s correspondent aux tilers ;

– les *OdtApiConnectionPoint*s correspondent aux ports de connexion.

Pour faciliter la tâche de l'utilisateur, les positions relatives des éléments UML dans les diagrammes (composants, propriétés, ports, ...) sont conservées.

De plus, pour la transformation de la représentation visuelle, en fonction des positions initiales dans les diagrammes composites du modèle avant la transformation, des nouvelles positions pour les éléments transformés sont calculées, avec la création de nouveaux diagrammes et le déplacement des éléments initiaux si nécessaire.

Exemple. La Figure 78 montre l'utilisation d'outil de refactoring dans l'éditeur Papyrus UML.

En fonction des éléments UML sélectionnés[4], le menu contextuel du diagramme composite (Subfig. 78a) propose les transformations disponibles. Des boites de dialogue seront utilisées pour la saisie des paramètres de la transformation (facteur de changement de pavage, dimensions, etc.).

[4] Des propriétés UML qui correspondent aux appels de sous-tâches.

La Subfig. 78b montre le résultat de la fusion de trois répétitions successives :

– les répétitions sont remplacées sur le diagramme par la répétition globale ;

– les sous-répétitions et le niveau inférieur de la composition sont placées sur des nouveaux diagrammes composites.

Observation. Les diagrammes composites (Figure 78) montrent uniquement des vues graphiques du modèle UML. Les vrais changements effectués par l'outil de refactoring sont au niveau UML et les diagrammes montrent ces changements pour apporter une vue visuelle du refactoring. Les transformations de haut-niveau peuvent être utilisées directement sur les objets UML de la vue *Outline*, avec les mêmes résultats exceptant les modifications de la représentation visuelle[5].

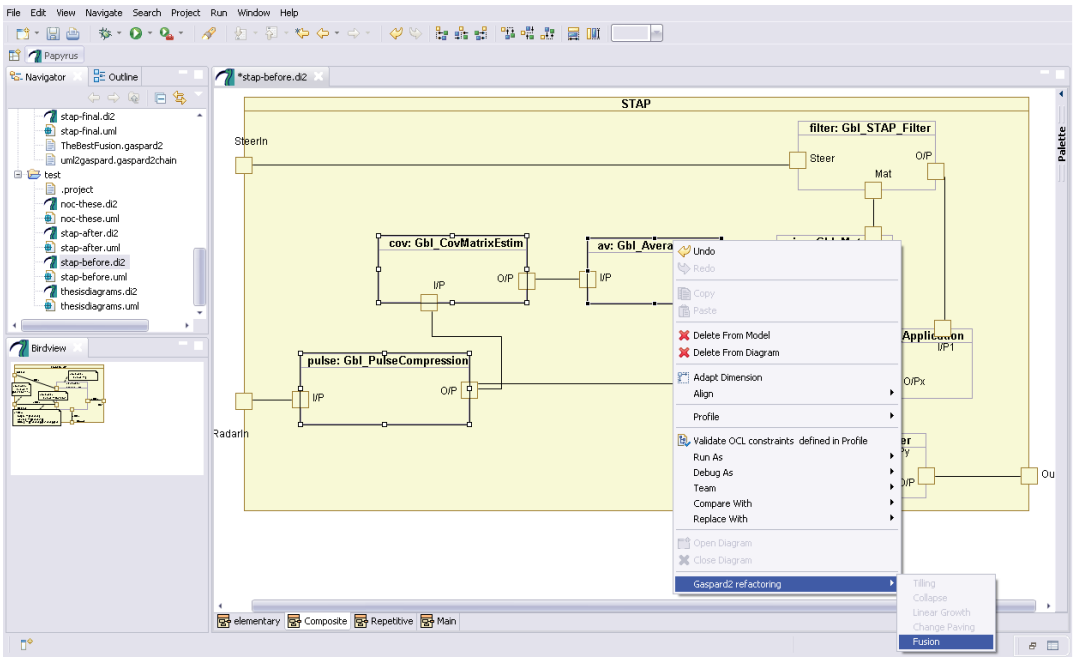
[5] Cette fonctionnalité peut être employée dans le cas où on ne dispose pas d'une vue graphique du modèle UML.

8.1.3 *Implémentation de l'interaction avec les dépendances interrépétitions*

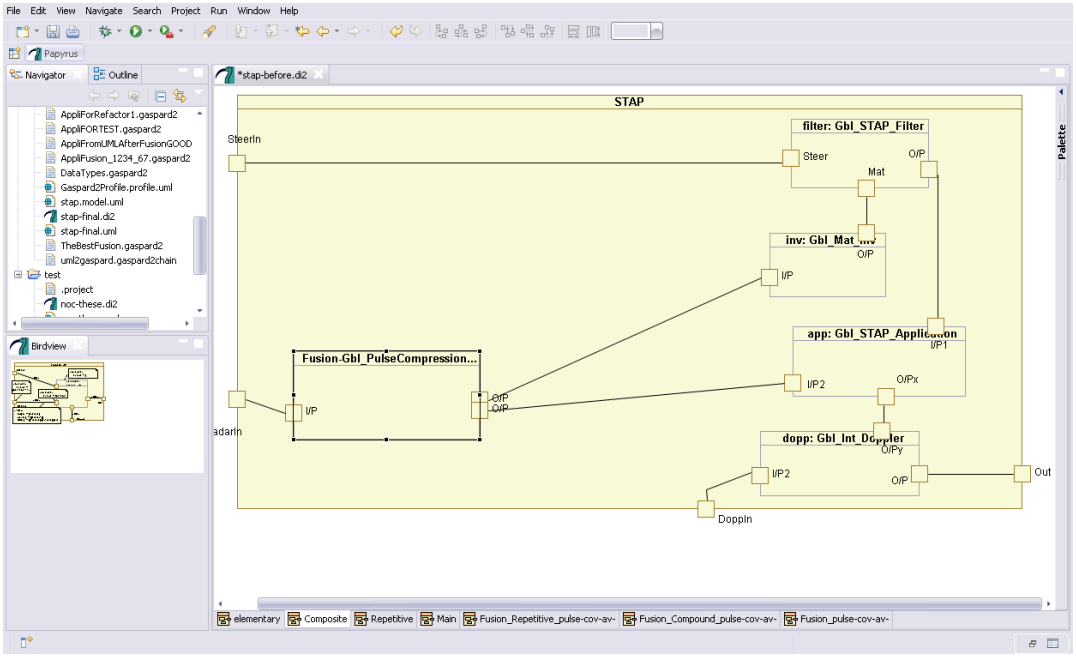
Pour gérer l'interaction des transformations de haut-niveau avec les dépendances interrépétitions, l'algorithme présenté dans le chapitre 6 a été ajouté à la boite d'outils de refactoring.

Au niveau de l'API, certains concepts ont été ajoutés pour exprimer les dépendances interrépétitions. Ces concepts sont illustrés sur la Figure 79.

Une étape de recalcul des dépendances a été ajoutée au calcul des transformations. Au niveau de l'API, après la traduction du résultat de la transformation ODT, la distribution des répétitions avant et



(a) Appel de la fusion



(b) Modèle résultat

FIG. 78: Utilisation d'outils de refactoring

après la transformation est identifiée et pour chaque dépendance inter-répétition avant la transformation, le/les dépendance(s) équivalente(s) sont calculée(s) par la résolution du système d'inéquations donné par l'Équation 6.12, sur la page 135.

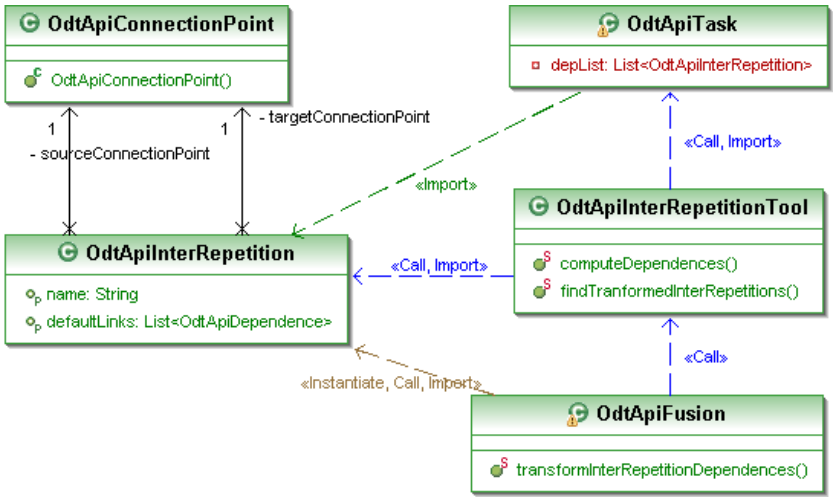


FIG. 79: Extension pour gérer les dépendances interrépétition

8.1.4 *Interfaçage avec l'atelier Ter@ops*

[6] Ce travail à été effectué dans le cadre du projet Ter@ops, http://www.teraopsemb.ief.u-psud.fr/.

Le projet Ter@ops[6] vise de proposer une solution apportant à la fois efficacité de traitement et souplesse d'emploi par une facilité de programmation pour de nombreuses applications de traitement intensif (imagerie, multimédia, etc.) dans le contexte des systèmes complexes « embarqués » requièrent de plus en plus de traitements intensifs et des performances temps réel garanties. L'approche Ter@ops est rendue possible par la mise en coopération des compétences systèmes associés aux compétences en plates-formes SoC, en compilation et architectures de traitement parallèles.

La première phase du projet Ter@ops inclut la mise en place d'une chaîne développement logiciel fondée sur l'approche ARRAY-OL, sur le langage C et sur un ensemble d'outils existants. Les outils participant au projet (APOTRES, GASPARD2, PIPS, SPEARDE, XLR8, GRAPHITE) doivent être fédérés autour de modèles communs de tâche, d'application, de machine cible et d'application placée.

L'interfaçage entre ces outils dans le cadre de l'atelier se fait par l'intermède de schémas XML correspondants aux modèles fournis par PIPS suite à l'analyse de code.

Pour l'intégration de GASPARD2 dans l'atelier, un outil d'import/export entre ce format et les modèles manipulés par GASPARD2 (UML + le profil MARTE) à été développé. Cela permet l'interfaçage avec les autres outils dans le projet et bénéficier du refactoring ARRAY-OL. La deuxième possibilité est l'exportation des modèles conçus en MARTE dans le format XML qui peuvent être utilisés par le reste de l'atelier. L'outil d'import/export est disponible à l'adresse http://gforge.inria.fr/frs/download.php/20222/fr.lifl.west.gaspard2.tools.convertor.pipsxml_0.2.2.jar.

8.2 PERSPECTIVES

Les transformations de haut-niveau data-parallèles permettent le refactoring de la spécification répétitive et facilitent l'exploration de l'espace de conception. L'intégration dans l'environnement de comod-

élisation GASPARD2 offre la possibilité de collaborer avec les autres fonctionnalités intégrées dans l'environnement.

Dans ce sens, l'utilisation des résultats des niveaux bas de la génération de code, la simulation ou l'exécution pour guider le refactoring peut offrir une solution pour augmenter les performances du code. Cette technique d'optimisation partage des concepts avec la compilation itérative. Les informations des niveaux bas peuvent être remontés au niveau spécification en utilisant les mécanismes de trace étudiés dans l'équipe par Aranega *et al.* [5].

Une autre direction d'évolution possible est l'implémentation et l'intégration dans l'environnement de stratégies de refactoring guidées par le concepteur, en fonction des gains en termes de taille mémoire, recalculs, granularité, etc.

8.3 CONCLUSIONS

Dans ce chapitre sont présentés les travaux d'implémentation dans le contexte de l'environnement GASPARD2, effectués pendant les années de thèse. Dans l'équipe DaRT de l'INRIA Lille-Nord Europe et de LIFL, une attention spéciale est mise sur l'intégration des différents travaux effectués dans l'équipe dans GASPARD2 et de les utiliser en commun.

L'intégration des outils de refactoring nous permet la validation des travaux théorétiques autour les techniques d'optimisation basées sur les transformations de haut-niveau, de bénéficier de résultats expérimentales remontés des niveaux simulation/exécution et d'explorer différentes stratégies de refactoring.

CONCLUSION GÉNÉRALE

BILAN

Les travaux présentés dans ce document s'inscrivent dans le cadre des recherches menées sur la modélisation et la conception des systèmes sur puce à hautes performances pour les applications de traitement intensif de signal à parallélisme massif. Dans cette thèse, nous nous sommes intéressés premièrement à la modélisation conjointe au plus haut niveau d'abstraction des applications massivement parallèles, des architectures répétées et du placement distribué des concepts applicatifs (calculs, données) sur des unités matérielles (processeurs, mémoires). Une spécification répétitive permet la modélisation visuelle du maximum de parallélisme disponible au niveau application, une représentation compacte des architectures répétées interconnectées et facilite l'exploration de l'espace de conception.

Nous avons présenté les différentes approches et différents modèles de calcul existants pour la spécification des applications de traitement intensif de signal à parallélisme massif, et nous avons vu que le langage de spécification Array-OL semblait bien adapté pour la description de l'aspect parallèle et multidimensionnel des données. Ses constructions d'accès par motifs réguliers permettent l'expression de l'ensemble des accès complexes (mais uniformes) qu'on peut rencontrer.

Pourtant, des règles de construction définies sur la spécification pour la rendre ordonnançable statiquement interdisent la construction des dépendances cycliques. Ensemble avec les principes de base du langage (assignation unique, la non-existence des variables globales, etc.), cela rendrait impossible d'exprimer des constructions d'état, essentielles dans un langage « complet » orienté flot de données. Pour surpasser cette limitation, nous avons proposé une extension du langage permettant l'expression des dépendances uniformes entre les répétitions de la même tâche. L'aspect uniforme de ces dépendances a un impact limité sur le parallélisme et des méthodes d'ordonnancement des répétitions à dépendances uniformes sont disponibles dans la littérature. Ce type de dépendance peut également être employé dans la modélisation compacte de topologies répétitives d'architectures matérielles.

Nous avons discuté des aspects liés à la comodélisation répétitive des applications, des architectures et du placement en MARTE. L'utilisation extensive des paquetages RSM et HRM de MARTE sert comme point d'entrée au niveau spécification de haut niveau de l'environnement Gaspard2 développé au sein de l'équipe DaRT où j'ai préparé cette thèse. Nous avons dédié un chapitre de ce document à la modélisation répétitive en MARTE pour Gaspard2 ; tous les concepts d'Array-OL sont disponibles dans le paquetage *Repetitive Structure Modeling*. Une spécification applicative en MARTE doit toujours être conforme au modèle de calcul Array-OL pour garder la propriété de la définition statique et nous avons montré comment une spécification en MARTE peut être réduite à celle du modèle de calcul Array-OL. La modélisation des architectures répétées est soumisse à moins de contraintes et cela permet l'expression compacte d'une large gamme de topologies répétitives.

L'expression complète du maximum de parallélisme au niveau de la spécification est essentielle pour pouvoir exploiter la régularité disponible dans les applications de traitement intensif de signal. Mais cela représente seulement le premier pas, tout dépend de comment cette spécification est exécutée dans le cadre des systèmes embarqués avec des unités de calcul parallèles.

Dans la deuxième partie de cette thèse, nous nous sommes intéressés au passage efficace de la spécification vers l'exécution par l'emploi des techniques d'optimisation de haut niveau permettant d'adapter une spécification aux contraintes de l'exécution, des ressources matérielles, de l'environnement, etc. Des outils de refactoring basés sur des transformations data-parallèles sont disponibles pour restructurer la spécification répétitive, en manipulant les concepts de granularité, hiérarchie, parallélisme.

L'ensemble des transformations déjà disponibles a été analysé et dans cette thèse nous avons proposé des extensions visant à étendre le domaine d'applicabilité et à mieux définir certaines propriétés liées aux transformations, à leur rôle et à leur impact sur la spécification.

Avec l'extension du langage Array-OL pour l'expression de dépendances uniformes, le problème de l'impact des dépendances sur le fonctionnement des transformations de haut niveau est apparu. Les calculs des transformations se rapportaient à un formalisme d'expression de dépendances entre des espaces multidimensionnels (l'ODT) et l'apparition des dépendances entre les éléments du même espace n'était pas exprimable sans des modifications au cœur du formalisme, avec des conséquences majeures à tous les niveaux du fonctionnement des transformations.

Pour gérer cette interaction entre les dépendances uniformes et les transformations de haut niveau sans devoir modifier radicalement tout le formalisme ODT, nous avons choisi une deuxième solution qui nous a permis de calculer les dépendances uniformes après une transformation de refactoring sans toucher au formalisme ODT. On s'appuyant sur quelques caractéristiques du fonctionnement des transformations et en mettant la condition que la sémantique de l'application et donc les dépendances de données doivent rester les mêmes après une transformation, nous avons réussi a formaliser et prouver un algorithme qui permet le calcul des dépendances uniformes sur les nouveaux espaces de répétition.

Avec l'objectif de valider en pratique l'ensemble de travaux théoriques autour le refactoring de haut niveau, l'implémentation et l'intégration dans l'environnement Gaspard2 était obligatoire. Cela nous a permis de vérifier l'exactitude et la validité des calculs de refactoring par des transformations Array-OL, ainsi que de l'algorithme de calcul des dépendances uniformes après de telles transformations.

L'intégration des outils de refactoring dans l'environnement Gaspard2 a ouvert la porte à des stratégies d'utilisation des transformations Array-OL. Nous avons proposé une telle stratégie pour la réduction des tailles des tableaux intermédiaires dans la spécification et nous avons illustré son fonctionnement sur la modélisation d'une application réelle de traitement de radar.

Néanmoins, le problème d'optimisation reste très complexe et est multicritères, favorisant des heuristiques ou des stratégies de refactoring aux algorithmes exacts. Une autre possibilité intéressante est une approche itérative d'optimisation, pilotée par des résultats remontant

des niveaux simulation/exécution/synthèse dans le cadre de l'environnement GASPARD2.

ANALYSE CRITIQUE

Mes travaux autour d'ARRAY-OL, les présentations que j'ai fait et des discutions autour du sujet m'ont amené à émettre plusieurs critiques sur ARRAY-OL et sur les langages de spécification en général.

Les langages de spécification facilitent la modélisation, mais sont la plupart du temps très restrictifs et très spécialisés. La sémantique de la description est parfois rigide et très proche au formalisme mathématique interne, notamment utilisée pour le calcul d'un ordonnancement statique. Un formalisme plus proche de l'utilisateur, plus général et plus flexible me semble plus approprié dans la perspective de l'utilisateur et des transformations vers la forme mathématique sont possibles derrière, transparentes à l'utilisateur.

J'ai constaté que certains principes d'ARRAY-OL restent souvent relativement obscurs, même pour des gens qui ont déjà travaillé dessus. Même si à la fin la construction des motifs d'accès uniformes est réduite à la maîtrise de la construction de tiler, la présence de plus de trois dimensions dans les formes multidimensionnelles rend difficile la visualisation des accès par le concepteur.

Une autre critique est en lien avec le formalisme ODT utilisé pour les calculs des transformations. Je considère que le formalisme est très restrictif et il est extrêmement laborieux d'ajouter des fonctionnalités. C'était le cas avec l'extension pour exprimer des dépendances uniformes ; le choix de contourner le formalisme ODT a beaucoup facilité le travail théorique et l'implémentation.

Une autre restriction est apportée par la contrainte de spécifier des valeurs numériques dans les tailles des tableaux et des tilers d'accès. La contrainte est principalement liée aux règles de construction pour garantir la définition statique, mais aussi par le calcul numérique des ODTs. Les règles de la définition statique peuvent être plus facilement adaptées pour permettre la spécification des paramètres, contrairement au formalisme ODT qui s'appuie sur des calculs numériques exacts (la transformation de fusion principalement).

PERSPECTIVES

Plusieurs directions d'évolutions futures peuvent être envisagées.

Il reste toujours des possibilités d'extension du langage ARRAY-OL pour améliorer encore son pouvoir d'expression. L'introduction des paramètres dans la spécification pourrait rendre une spécification plus générale. L'adaptation des templates UML pour la modélisation des composants paramétrables dans GASPARD2 a été étudiée par de Moura *et al.* en [26]. Des aspects dynamiques peuvent être envisagés avec l'observation que cela signifierait la perte de la définition statique et de l'utilisation des outils de refactoring. Une solution de compromis pourrait être représentée par l'introduction des aspects dynamiques uniquement à certains niveaux de la hiérarchie et en gardant des aspects statiques aux niveaux locaux.

Des évolutions des transformations de haut niveau peuvent être développées aussi, mais il faut dépasser les limitations introduites par

les ODT. Le cas de la fusion avec de multiples tableaux intermédiaires reste toujours sans solution adéquate.

Avec l'intégration des outils de refactoring dans l'environnement de comodélisation GASPARD2 ouvre des nouvelles directions de développement. Il s'agit des travaux de conception de nouvelles stratégies d'optimisation pilotées par des résultats remontant des niveaux de simulation/exécution/synthèse par des mécanismes de trace.

Quatrième partie

ANNEXES

A

APPLICATION DE FENÊTRES GLISSANTES EN ARRAY-OL

Dans la Figure 2.5.4, nous avons vu que, sans l'utilisation des concepts de *Reshape* et de *Lien par défaut*, on ne peut pas exprimer avec Array-OL exactement la même application que celle décrite en WSDF. On peut argumenter que l'utilisation des *unions des jetons virtuels* ne se justifie pas entièrement ; la modélisation des images d'une union comme une seule « entité » depuis le début peut résoudre le problème.

L'extension des frontières non-cyclique est aussi possible en utilisant le concept de lien par défaut, emprunté de la spécification des dépendances uniformes. Dans le contexte des fenêtres glissantes, le concept a une signification similaire, il fourni des éléments avec des valeurs par défaut dans le cas où l'accès se fait à l'extérieur du tableau, en substitution à l'accès cyclique.

En utilisant les concepts de reshape et de lien par défaut, on peut modéliser complètement l'application :

- le reshape pour redéfinir la forme des tableaux, un flot d'images avec un groupe de 2*2 images successives et traité comme un bloc continu est transformé dans un flot de blocs, chaque bloc contenant 2*2 images ;
- le lien par défaut exprime l'extension des frontières avec une valeur par défaut.

La Figure 80 montre cette spécification dans une représentation Array-OL et la Figure 81 montre la même spécification en MARTE RSM.

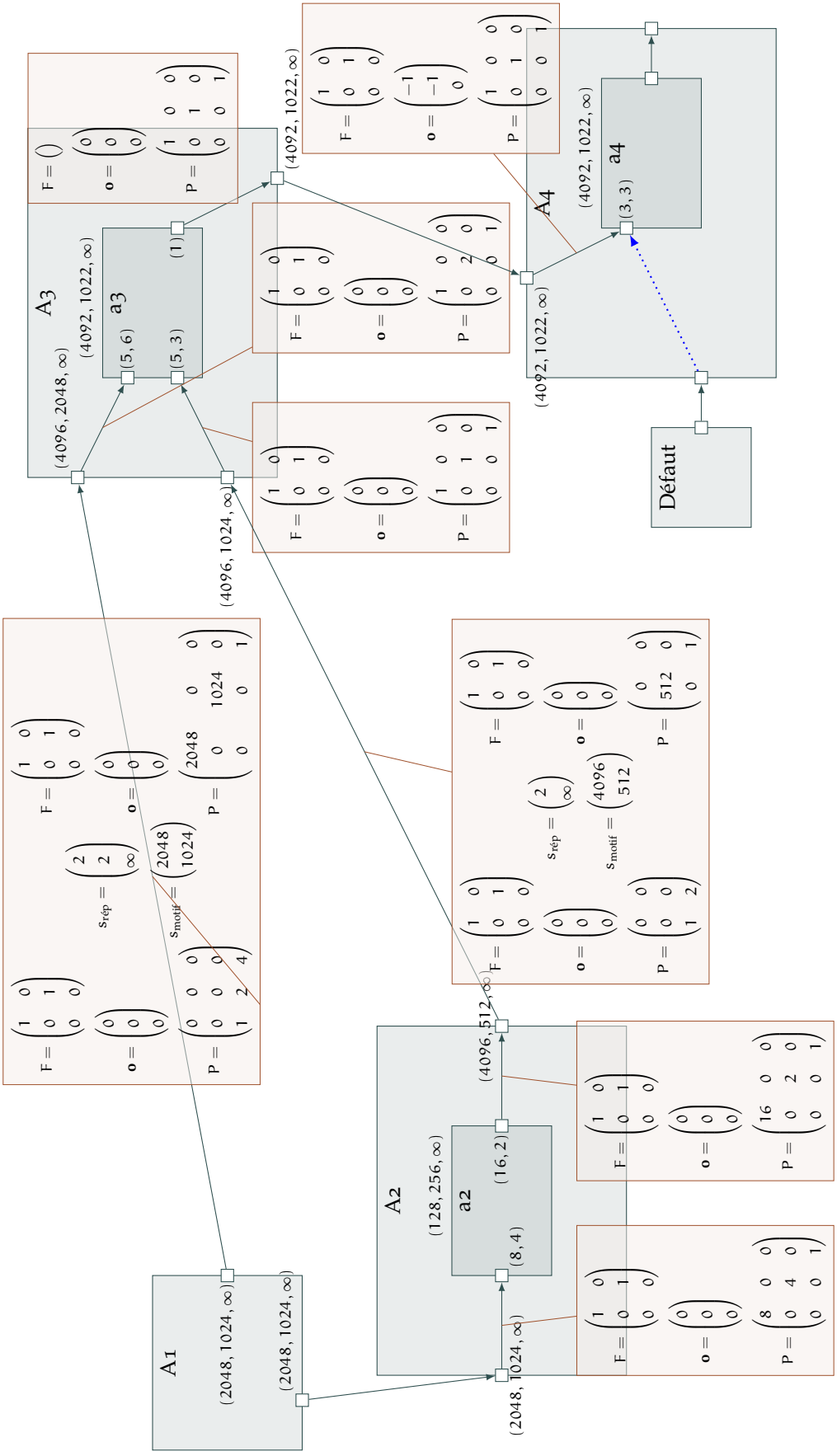


FIG. 80: Fenêtres glissantes en ARRAY-OL en utilisant des *Reshapes* et des *Liens par défaut*

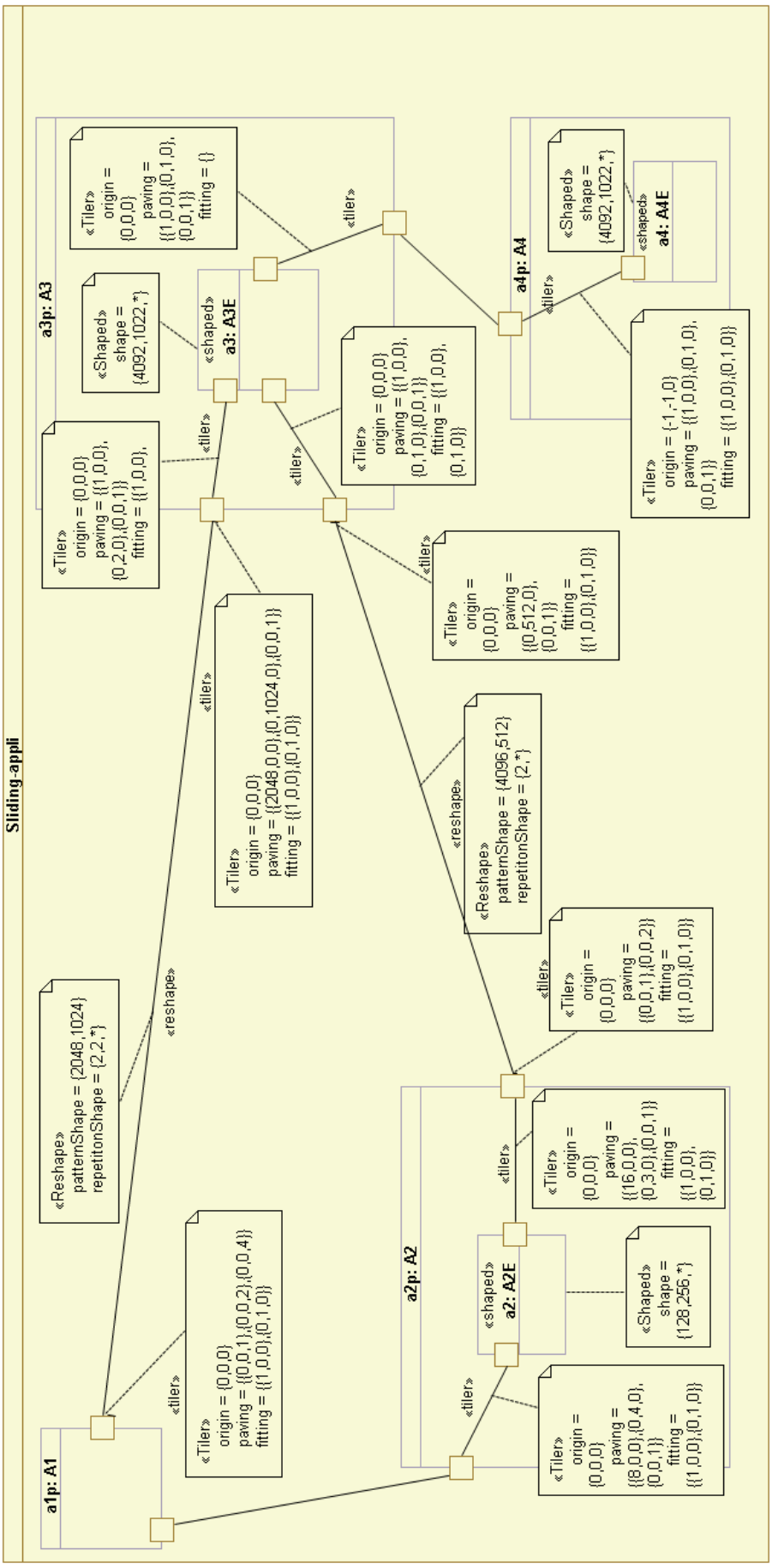


FIG. 81: Fenêtres glissantes en MARTE

B

MODÉLISATION D'UNE ARCHITECTURE NOC EN MARTE

Figure 82 montre une topologie de NoC qui a une forme assez régulière. Nous allons voir comment nous pouvons modéliser ce type d'architecture sous une forme compacte et facilement modifiable. Nous allons utiliser des valeurs numériques dans la description répétitive mais ces valeurs peuvent être changées facilement pour exprimer des architectures avec une structure similaire mais nombre de nœds différents.

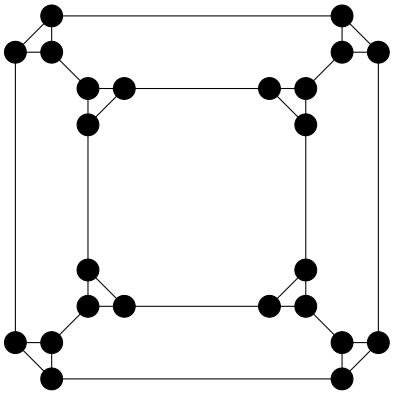
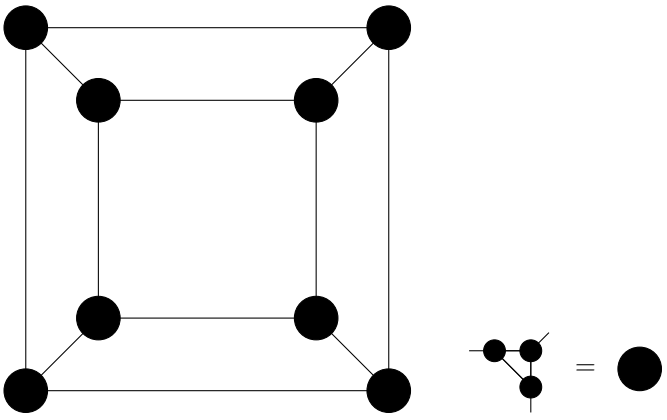


FIG. 82: Topologie de NoC

Même si la structure du NoC sur la Figure 82 n'est pas parfaitement régulière, nous allons voir comment les concepts répétitives de MARTE RSM peuvent être utilisés pour l'expression des architectures où la régularité n'est pas directement exprimable par les concepts répétitifs de base.

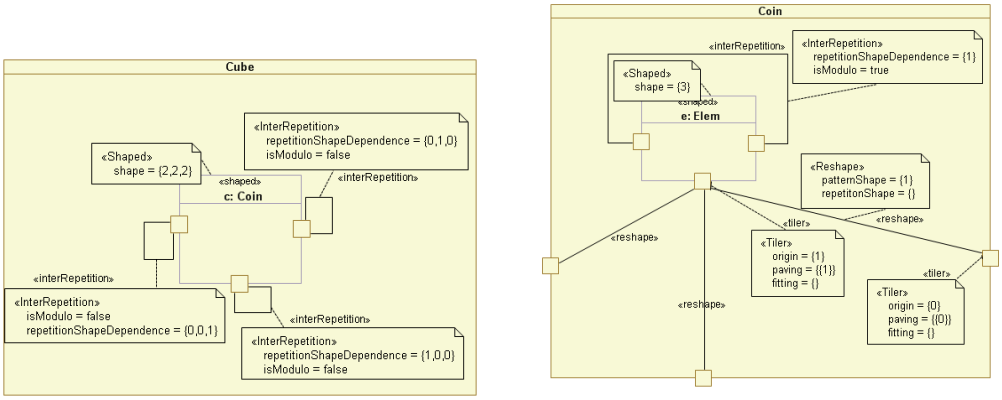


(a) Topologie de cube (b) Topologie du coin

FIG. 83: Partage hiérarchique du NoC

Si nous considérons chaque groupe de trois nœds comme une entité unique (*cf* Subfig. 83b), nous nous retrouvons avec une topologie de

cube (*cf* Subfig. 83a) qui peut être représentée facilement par un espace de répétition trois-dimensionnel et des dépendances inter-répétition. Les nœds des coins peuvent être représentés par une répétition au deuxième niveau de la hiérarchie.



(a) Cube (b) Coin

FIG. 84: Partage hiérarchique du NoC

La Figure 84 montre la modélisation en MARTE RSM du NoC, la Subfig. 84b le niveau des trois nœuds du coin et la Subfig. 84a le niveau du cube :

- tous les ports ont une forme (*shape*) vide correspondant à un seul élément ({}) ;
- les trois reshapes de la Subfig. 84b expriment la séparation d'un port avec une forme de ({3}[1]) en trois ports, par l'utilisation des origines différentes pour les origines des tilers sources des reshapes.
  Observation. Une seule reshape est montrée sur le dessin, les autres sont identiques avec la différence de l'origine du tiler source : {0} pour le premier élément, {1} pour le deuxième élément et {2} pour le troisième élément.
- les dépendances inter-répétition sur la Subfig. 84a expriment des liens non cycliques pour chaque dimension, sur le port qui correspond à cette dimension ; la dimension est donnée par la valeur non-nulle du vecteur de dépendance.

[1] Parce que les connecteurs ne sont pas des tilers, la forme du port est donnée par la concaténation avec l'espace de répétition, *cf* les règles des tâches répétitives-composées, sous-section 3.6.2.

BIBLIOGRAPHIE

[1] Cloog home page. http://www.cloog.org. (Cité à la page 70.)

[2] A. Koudri et al : Using MARTE in the MOPCOM SoC/SoPC Co-Methodology. *In MARTE Workshop at DATE'08*, 2008. (Cité à la page 78.)

[3] Abdelkader Amar, Pierre Boulet et Philippe Dumont : Projection of the Array-OL specification language onto the Kahn process network computation model. *In International Symposium on Parallel Architectures, Algorithms, and Networks*, Las Vegas, Nevada, USA, décembre 2005. (Cité à la page 60.)

[4] C. Ancourt, D. Barthou, C. Guettier, F. Irigoin, B. Jourdan et J. Mattioli : Automatic data mapping of signal processing applications. *In Application Spec. Array Processors*, pages 350–362, Zurich, Switzerland, juillet 1997. (Cité à la page 88.)

[5] Vincent Aranega, Jean-Marie Mottu, Anne Etien et Jean-Luc Dekeyser : Traceability mechanism for error localization in model transformations. *In 4th International Conference on Software and Data Technologies (ICSOFT 2009)*, Sofia, Bulgaria, July 2009. (Cité à la page 165.)

[6] Isabelle Attali, Denis Caromel, Yung syau Chen, Jean luc Gaudiot et Andrew L. Wendelborn : A formal semantics for sisal arrays. *In Proc. Joint Conf. Infor. Sci.*, septembre 1995. (Cité aux pages 11 et 13.)

[7] David F. Bacon, Susan L. Graham et Oliver J. Sharp : Compiler transformations for high-performance computing. Rapport technique, Berkeley, CA, USA, 1993. (Cité aux pages 90 et 92.)

[8] Marcus Bednara et Jürgen Teich : Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms. *The Journal of Supercomputing*, 26(2):149–165, septembre 2003. (Cité à la page 78.)

[9] Rabie Ben Atitallah : *Modèles et simulation de systèmes sur puce multiprocesseurs – Estimation des performances et de la consommation d'énergie*. Thèse de doctorat, „ France, décembre 2007. (Cité à la page 70.)

[10] Rabie Ben Atitallah, Pierre Boulet, Arnaud Cuccuru, Jean-Luc Dekeyser, Antoine Honoré, Ouassila Labbani, Sébastien Le Beux, Philippe Marquet, Éric Piel, Julien Taillard et Huafeng Yu : Gaspard2 uml profile documentation. Rapport technique 0342, septembre 2007. URL http://hal.inria.fr/inria-00171137/en. (Cité à la page 71.)

[11] L. Benini et G. De Micheli : Networks on chips : a new SoC paradigm. *Computer*, 35(Issue : 1):70–78, Jan 2002. (Cité à la page 8.)

[12] Bishnupriya. Bhattacharya, Md.). Dept. of Electrical University of Maryland (College Park et Computer Engineering. : Parameterized modeling and scheduling for dataflow graphs /. 1999. URL http://worldcat.org/oclc/44599249. (Cité à la page 16.)

[13] G. Bilsen, M. Engels, R. Lauwereins et J.A. Peperstraete : Cyclo-static data flow. *In International Conference on Acoustics, Speech, and Signal Processing.*, Detroit, MI, USA, mai 1995. (Cité aux pages 11 et 16.)

[14] Pierre Boulet : Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing. Rapport technique RR-6467, mars 2008. URL http://hal.inria.fr/inria-00261178/en/. (Cité aux pages 11, 28, 38, 39, et 40.)

[15] Pierre Boulet, Philippe Marquet, Éric Piel et Julien Taillard : Repetitive Allocation Modeling with MARTE. *In Forum on specification and design languages (FDL'07)*, Barcelona, Spain, septembre 2007. Invited Paper. (Cité à la page 81.)

[16] Pierre Boulet et Xavier Redon : SPPoC : manipulation automatique de polyèdres pour la compilation. 20(8):1019–1048, 2001. (Cité à la page 40.)

[17] L. Cai et D. Gajski : Transaction level modeling : an overview. *In CODES+ISSS'03*, 2003. (Cité à la page 70.)

[18] Michael J. Chen et Edward A ; Lee : Design and implementation of a multidimensional synchronous dataflow environment. *In 1995 Proc. IEEE Asilomar Conf. on Signal, Systems, and Computers*, 1995. (Cité aux pages 11, 16, et 18.)

[19] Arnaud Cuccuru : *Modélisation unifiée des aspects répétitifs dans la conception conjointe logicielle/matérielle des systèmes sur puce à hautes performances*. Thèse de doctorat, novembre 2005. URL http://www.lifl.fr/west/publi/Cucc05phd.pdf. (Cité à la page 71.)

[20] DaRT Team : Graphical Array Specification for Parallel and Distributed Computing (GASPARD2). http://www.gaspard2.org/, 2009. (Cité à la page 70.)

[21] Alain Darte : De l'organisation des calculs dans les codes répétitifs. Habilitation à diriger des recherches, École Normale Supérieure de Lyon, décembre 1999. (Cité à la page 100.)

[22] Alain Darte et Yves Robert : Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems*, 5(8): 814–822, 1994. (Cité à la page 61.)

[23] Alain Darte, Yves Robert et Frédéric Vivien : *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000. http://www.birkhauser.com/detail.tpl?isbn=0817641491. (Cité à la page 90.)

[24] Florent de Dinechin, Patrice Quinton et Tanguy Risset : Structuration of the alpha language. *In Programming Models for Massively Parallel Computers*, pages 18–24, Berlin, Germany, octobre 1995. (Cité aux pages 11 et 12.)

[25] Eddy De Greef, Francky Catthoor et Hugo De Man : Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Comput.*, 23(12):1811–1837, 1997. ISSN 0167-8191. (Cité aux pages 111 et 127.)

[26] Cesar de MOURA, Julien TAILLARD, Frédéric GUYOMARC'H et Cedric DUMOULIN : Adaptation des Templates UML pour la modélisation de composants paramétrables : application à Gaspard2 . *In 4èmes Jounées sur l'Ingénierie Dirigée par les Modèles (IDM 08)*, Mulhouse, France, juin 2008. (Cité à la page 169.)

[27] Alain DEMEURE : Les ODT : Propositions de notation pour décrire des opérateurs de distribution de tableaux. Rapport technique, Thomson Marconi Sonar, Sophia-Antipolis, France, 1998. (Cité à la page 100.)

[28] Alain DEMEURE et Yannick DEL GALLO : An Array Approach for Signal Processing Design. *In Sophia-Antipolis conference on Micro-Electronics (SAME 98)*, France, octobre 1998. (Cité à la page 11.)

[29] Alain DEMEURE, Anne LAFARGE, Emmanuel BOUTILLON, Didier ROZZONELLI, Jean-Claude DUFOURD et Jean-Louis MARRO : Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. *In Gretsi*, Juan-Les-Pins, France, septembre 1995. (Cité à la page 11.)

[30] Tata Research DEVELOPMENT et Design CENTRE : Modelmorf – a model transformer. http://www.tcs-trddc.com/ModelMorf. (Cité à la page 64.)

[31] M. DINCBAS, P. Van HENTENRYCK, H. SIMONIS, A. AGGOUN, T. GRAF et F. BERTHIER : The constraint logic programming language chip. *In International Conference on Fifth Generation Computer Systems*, pages 693–264, 1988. (Cité à la page 88.)

[32] Philippe DUMONT : *Spécification Multidimensionnelle pour le traitement du signal systématique*. Thèse de doctorat, „ décembre 2005. (Cité aux pages 4, 100, 105, 108, 109, 114, 115, 118, 120, 124, et 159.)

[33] Philippe DUMONT et Pierre BOULET : Another multidimensional synchronous dataflow : Simulating Array-OL in ptolemy II. Rapport technique RR-5516, mars 2005. URL http://www.inria.fr/rrrt/rr-5516.html. (Cité à la page 42.)

[34] Jean-Marie FAVRE : Foundations of model (driven) (reverse) engineering : Models – episode i : Stories of the fidus papyrus and of the solarus. *In* Jean BEZIVIN et Reiko HECKEL, éditeurs : *Language Engineering for Model-Driven Software Development*, numéro 04101 *in* Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http ://drops.dagstuhl.de/opus/volltexte/2005/13> [date of citation : 2005-01-01]. (Cité à la page 64.)

[35] P. FEAUTRIER : Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988. (Cité à la page 53.)

[36] Paul FEAUTRIER : Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991. URL citeseer.ist.psu.edu/feautrier91dataflow.html. (Cité à la page 93.)

[37] Paul Feautrier : Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 21(1):23–53, 1991. (Cité à la page 100.)

[38] Antoine Fraboulet : *Optimisation de la mémoire et de la consommation des systèmes multimédia embarque*. Thèse de doctorat, LIP, novembre 2001. (Cité à la page 93.)

[39] Grigori Fursin et Albert Cohen : Building a practical iterative interactive compiler. *In 1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), colocated with HiPEAC 2007 conference*, January 2007. (Cité à la page 93.)

[40] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Böhm, Walid Najjar et Patrick Miller : *Compiler optimizations for scalable parallel systems : languages, compilation techniques, and run time systems*, chapitre The Sisal project : real world functional programming, pages 45–72. Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 3-540-41945-4. (Cité aux pages 11 et 13.)

[41] Sylvain Girbal : *Optimisation dapplications - Composition de transformations de programme : modele et outils*. Thèse de doctorat, University Paris 11, Orsay, France, September 2005. (Cité à la page 93.)

[42] Susanne Graf, Oystein Haugen, Iulian Ober et Bran Selic : Modeling and analysis of real-time and embedded systems : workshop overview. *In Modeling and Analysis of Real-time and Embedded Systems*, 2005. (Cité à la page 78.)

[43] Pascal Van Hentenryck, Helmut Simonis et Mehmet Dincbas : Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1-3):113–159, 1992. URL `citeseer.ist.psu.edu/vanhentenryck92constraint.html`. (Cité à la page 88.)

[44] High Performance Fortran Forum : High performance fortran language specification version 2.0. Rapport technique, Department of Computer Science, Rice University, 1997. (Cité à la page 15.)

[45] American National Standards Institute : American national standard programming language, fortran - extended : Ansi x3.198-1992 : Iso/iec 1539 : 1991 (e). Rapport technique, 1992. (Cité à la page 15.)

[46] A. Jerraya et W. Wolf, éditeurs. *Multiprocessor Systems-on-Chip*. Elsevier Morgan Kaufmann, San Francisco, California, 2005. (Cité à la page 8.)

[47] Jürgen Teich Joachim Keinert, Christian Haubelt : Windowed synchronous data flow. Technical Report Co-Design-Report 02, 2005, Department of Computer Science 12, Hardware-Software-Co-Design, University of Erlangen-Nuremberg, Am Weichselgarten 3, D-91058 Erlangen, Germany, 2005. (Cité aux pages 11, 16, et 24.)

[48] Frédéric Jouault et Ivan Kurtev : Transforming Models with ATL. *In Proceedings of the Model Transformation in Practice Workshop*, Montego Bay, Jamaica, octobre 2005. (Cité à la page 64.)

[49] Richard M. KARP, Raymond E. MILLER et Shmuel WINOGRAD : The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, juillet 1967. (Cité à la page 12.)

[50] Joachim KEINERT, Christian HAUBELT et Jürgen TEICH : Modeling and analysis of windowed synchronous algorithms. *In International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages III–892– III–895, 2006. (Cité aux pages 11, 16, et 24.)

[51] Ken KENNEDY et John R. ALLEN : *Optimizing compilers for modern architectures : a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0. (Cité à la page 90.)

[52] Ken KENNEDY et Kathryn S. MCKINLEY : Maximizing loop parallelism and improving data locality via loop fusion and distribution. *In 1993 Workshop on Languages and Compilers for Parallel Computing*, numéro 768, pages 301–320, Portland, Ore., 1993. Berlin : Springer Verlag. URL citeseer.ist.psu.edu/kennedy94maximizing.html. (Cité à la page 92.)

[53] Toru KISUKI, Peter M. W. KNIJNENBURG, Michael F. P. O'BOYLE, François BODIN et Harry A. G. WIJSHOFF : A feasibility study in iterative compilation. *In ISHPC '99 : Proceedings of the Second International Symposium on High Performance Computing*, pages 121–132, London, UK, 1999. Springer-Verlag. ISBN 3-540-65969-2. (Cité à la page 93.)

[54] L. BASS AND P. CLEMENTS AND R. KAZMAN : Software Architecture In Practice. *Addison Wesley*, 1998. (Cité à la page 77.)

[55] Ouassila LABBANI, Jean-Luc DEKEYSER, Pierre BOULET et Éric RUTTEN : UML2 profile for modeling controlled data parallel applications. *In FDL'06 : Forum on Specification and Design Languages*, Darmstadt, Germany, septembre 2006. (Cité à la page 149.)

[56] Ouassila LABBANI, Jean-Luc DEKEYSER, Pierre BOULET et Éric RUTTEN : Introducing control in the gaspard2 data-parallel metamodel : Synchronous approach. *International Workshop MARTES : Modeling and Analysis of Real-Time and Embedded Systems (in conjunction with 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005)*, octobre 2005. (Cité aux pages 29, 111, et 149.)

[57] Leslie LAMPORT : The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, 1974. ISSN 0001-0782. (Cité à la page 61.)

[58] Sébastien LE BEUX : *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. Thèse de doctorat, „ France, décembre 2007. (Cité à la page 70.)

[59] Hervé LE VERGE, Christophe MAURAS et Patrice QUINTON : The alpha language and its use for the design of systolic arrays. *The Journal of VLSI Signal Processing*, 3(3):173–182, septembre 1991. (Cité aux pages 11 et 12.)

[60] E. A. Lee et D. G. Messerschmitt : Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, janvier 1987. (Cité aux pages 11, 16, et 18.)

[61] E. A. Lee et D. G. Messerschmitt : Synchronous Data Flow. *Proc. of the IEEE*, 75(9):1235–1245, septembre 1987. (Cité aux pages 11 et 16.)

[62] Edward A. Lee : Multidimensional streams rooted in dataflow. *In Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, janvier 1993. North-Holland. (Cité aux pages 11, 16, 18, et 20.)

[63] Tom Mens, Krzysztof Czarnecki et Pieter Van Gorp : A taxonomy of model transformations. *In* Jean Bezivin et Reiko Heckel, éditeurs : *Language Engineering for Model-Driven Software Development*, numéro 04101 *in* Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/11. (Cité à la page 64.)

[64] S. Mohanty, V. K. Prasanna, S. Neema et J. Davis : Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *In Conference on Languages, compilers and tools for embedded systems*, Berlin, Germany, 2002. (Cité à la page 78.)

[65] G. E. Moore : Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965. URL http://dx.doi.org/10.1109/JPROC.1998.658762. (Cité à la page 1.)

[66] A. Mozipo, D. Massicotte, P. Quinton et T. Risset : Automatic synthesis of a parallel architecture for kalman filtering using mmalpha. *In International Conference on Parallel Computing in Electrical Engineering (PARELEC 98)*, pages 201–206, Bialystok, Poland, 1998. (Cité à la page 78.)

[67] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer et Jean-Marc Jézéquel : On executable meta-languages applied to model transformations. *In Model Transformations In Practice Workshop*, Montego Bay, Jamaica, octobre 2005. (Cité à la page 64.)

[68] Praveen K. Murthy et Edward A. Lee : A generalization of multidimensional synchronous dataflow to arbitrary sampling lattices. Research report, Electronics Research Laboratory, mars 1995. (Cité à la page 23.)

[69] Praveen K. Murthy et Edward A. Lee : Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50 (8):2064–2079, août 2002. (Cité aux pages 11, 16, 20, 21, 22, 23, 24, et 43.)

[70] Praveen Kumar Murthy : *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. Thèse de doctorat, University of California, Berkeley, CA, 1996. (Cité aux pages 11, 16, et 20.)

[71] OBJECT MANAGEMENT GROUP : A UML profile for MARTE, 2007. http://www.omgmarte.org. (Cité à la page 65.)

[72] OBJECT MANAGEMENT GROUP, INC., éditeur. *UML 2 Infrastructure (Final Adopted Specifcation)*. http://www.omg.org/cgi-bin/doc?ptc/2003-09-15, septembre 2003. (Cité à la page 64.)

[73] OBJECT MANAGEMENT GROUP, INC., éditeur. *(UML) Profile for Schedulability, Performance, and Time Version 1.1*. http://www.omg.org/technology/documents/formal/schedulability.htm, janvier 2005. (Cité à la page 77.)

[74] OBJECT MANAGEMENT GROUP INC. : Final adopted omg sysml specification. http ://www.omg.org/cgi-bin/doc ?ptc/06-0504, mai 2006. (Cité à la page 77.)

[75] OBJECT MANAGEMENT GROUP, INC. : MOF Query / Views / Transformations. http://www.omg.org/docs/ptc/07-07-07.pdf, juillet 2007. OMG paper. (Cité à la page 65.)

[76] OMG : UML profile for Schedulability, Performance and Time (SPT), Version 1.1, 2005. (Cité à la page 77.)

[77] OMG : UML profile for System on Chip (SoC), Version 1.0.1, 2006. (Cité à la page 77.)

[78] Éric PIEL : *Ordonnancement de systèmes parallèles temps-réel, De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles*. Thèse de doctorat, „ France, décembre 2007. (Cité à la page 70.)

[79] Éric PIEL, Rabie Ben ATTITALAH, Philippe MARQUET, Samy MEFTALI, Smaïl NIAR, Anne ETIEN, Jean-Luc DEKEYSER et Pierre BOULET : Gaspard2 : from MARTE to SystemC simulation. *In Modeling and Analyzis of Real-Time and Embedded Systems with the MARTE UML profile DATE'08 Workshop*, mars 2008. URL http://www2.lifl.fr/MARTEworkshop/. (Cité aux pages 65 et 70.)

[80] PROMARTE PARTNERS : UML Profile for MARTE, Beta 2. http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf, juin 2008. (Cité à la page 65.)

[81] E DE RECHERCHE, ET EN AUTOMATIQUE, DOMAINE DE VOLUCEAU, Christine EISENBEIS, Christine EISENBEIS, William JALBY, William JALBY, Daniel WINDHEISER, Daniel WINDHEISER, Francois BODIN et Francois BODIN : A strategy for array management in local memory, 1990. (Cité à la page 111.)

[82] E. RICCOBENE, P. SCANDURRA, A. ROSTI et S. BOCCHIO : A model-driven design environment for embedded systems. *In Proceedings of the 43rd annual Design Automation Conference (DAC '06)*, pages 915–918. ACM, 2006. (Cité à la page 77.)

[83] L. RIOUX, T. SAUNIER, S. GERARD, A. RADERMACHER, R. DE SIMONE, T. GAUTIER, Y. SOREL, J. FORGET, J.-L. DEKEYSER, A. CUCCURU, C. DUMOULIN et C. ANDRE : MARTE : A new profile RFP for the modeling and analysis of real-time embedded systems. *In UML-SoC'05, DAC 2005 Workshop UML for SoC Design*, Anaheim, CA, juin 2005. (Cité à la page 65.)

[84] Sven-Bodo Scholz : Single assignment c : efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13 (6):1005–1059, 2003. ISSN 0956-7968. (Cité aux pages 11 et 14.)

[85] Julien Soula : *Principe de Compilation d'un Langage de Traitement de Signal*. Thèse de doctorat, „ décembre 2001. (Cité aux pages 4, 100, 105, 109, et 119.)

[86] Julien Taillard : *Une approche orientée modèle pour la parallélisation d'un code de calcul éléments finis*. Thèse de doctorat, Université des Sciences et Technologies de Lille, feb 2009. in french. (Cité à la page 70.)

[87] William Thies, Michal Karczmarek et Saman Amarasinghe : StreamIt : A language for streaming applications. *In Compiler Construction : 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002. Proceedings.*, volume 2304/2002 de *Lecture Notes in Computer Science*, pages 49–84. Springer Berlin / Heidelberg, 2002. (Cité aux pages 11 et 14.)

[88] INRIA Triskell : Kermeta. http://www.kermeta.org/. (Cité à la page 64.)

[89] Jan Vanhoof, Ivo Bolsens, Karl Van Rompaey, Gert Goossens et Hug De Man : *High-Level Synthesis for Real-Time Digital Signal Processing*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0792393139. (Cité à la page 111.)

[90] W. Cesario et al : Component-Based Design Approach for Multicore SoCs. *Design Automatic Conference, DAC'02*, 00:789, 2002. (Cité à la page 78.)

[91] P. Wauters, M. Engels, R. Lauwereins et J.A. Peperstraete : Cyclo-dynamic dataflow. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:0319, 1996. (Cité à la page 16.)

[92] Michael Joseph Wolfe : *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0805327304. (Cité à la page 90.)

[93] Huafeng Yu : *A MARTE-Based Reactive Model for Data-Parallel Intensive Processing : Transformation Toward the Synchronous Model*. Thèse de doctorat, Lille, France, November 2008. (Cité à la page 70.)

[94] Hans Zima et Barbara Chapman : *Supercompilers for parallel and vector computers*. ACM, New York, NY, USA, 1991. ISBN 0-201-17560-6. (Cité à la page 90.)