Vers des transformations d'applications à parallélisme de données en équations synchrones

Huafeng Yu, Abdoulaye Gamatié, Eric Rutten, Pierre Boulet, Jean-Luc Dekeyser

RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

Vers des transformations d'applications à parallélisme de données en équations synchrones

Huafeng Yu, Abdoulaye Gamatié, Éric Rutten, Pierre Boulet, et Jean-Luc Dekeyser

INRIA Futurs/LIFL
Synergie Park, 6bis avenue Pierre et Marie Curie, 59260 Lezennes, France
{yu, gamatie, rutten, boulet, dekeyser}@lifl.fr

Résumé
Ce papier présente les premiers résultats d'une étude concernant la transformation d'applications à parallélisme de données en équations synchrones. Les applications considérées sont exprimées à l'aide du métamodèle GASPARD qui étend le langage ARRAY-OL, dédié aux applications de traitement de données intensives. Le principe général des transformations envisagées est exposé ainsi que les idées de mise en œuvre. Les modèles synchrones résultants permettent d'aborder plusieurs questions liées à la validation formelle, par exemple, vérification de propriétés de synchronisabilité, de latence, etc, en utilisant les outils et techniques formels offerts par la technologie synchrone. Ils permettent ainsi l'accès à des fonctionnalités complémentaires avec celles de l'environnement associé à GASPARD, qui propose une méthodologie de conception conjointe matériel/logiciel de systèmes intégrés sur puce. Les transformations suivront une approche d'Ingénierie dirigée par les modèles (IDM/MDE). Des perspectives sont mentionnées concernant l'introduction d'automates de contrôle au sein des modèles obtenus.

Mots-clés : parallélisme de données, modélisation, modèle synchrone flot de données.

1. Introduction

Le traitement et l'analyse de données en masse représente un enjeu de plus en plus important dans les systèmes embarqués. Ces applications manipulent la plupart du temps des structures de données multidimensionnelles régulières. Des exemples typiques sont le multimedia, par exemple, la télévision à haute définition, l'imagerie médicale ; le traitement de signaux radar ou sonar, les télécommunications, etc. Pour ces applications, les approches de conception fortement recherchées sont celles qui fournissent aux utilisateurs des concepts bien adaptés pour représenter les manipulations de données, et des techniques qui garantissent les propriétés requises à leur mise en œuvre correcte.

Un objectif majeur de l'environnement GASPARD est d'apporter une réponse adéquate à cette attente. Son formalisme de spécification, appelé ARRAY-OL [8], a été défini initialement par Thomson Marconi Sonar, dans un contexte industriel. Il permet de décrire des applications de traitement de données intensives en manipulant des structures de données multidimensionnelles régulières. Les aspects traités sont purement fonctionnels. ARRAY-OL ne permet pas d'aborder des aspects non fonctionnels tels que les contraintes temporelles pouvant être imposées par l'environnement, ou bien le contrôle induit par différentes configurations ou modes de fonctionnement lors de certains calculs.

Dans ce papier, nous proposons donc un modèle synchrone d'applications de traitement de données intensives, basés sur l'extension d'ARRAY-OL dans GASPARD. Les structures de données manipulées sont des tableaux multidimensionnels, toriques, et potentiellement infinis. Les spécifications fournies par ARRAY-OL expriment des dépendances de données basées uniquement sur les valeurs, c'est-à-dire de *vraies* dépendances de données, exhibant ainsi un ordre minimal d'exécution. Elles adoptent un style à assignation unique. De plus, elles sont indépendantes des détails d'architectures. Toutes ces caractéristiques confèrent à ARRAY-OL une expressivité puissante pour la manipulation de données dans

les applications de traitement de données intensives. Le modèle que nous proposons vise à fournir d'une part, la même expressivité que les descriptions GASPARD et d'autre part, la possibilité de traiter le contrôle et les contraintes non fonctionnelles dans les applications. Il permet notamment d'explorer, à des niveaux élevés, différents raffinements possibles de descriptions GASPARD initiales vis-à-vis des contraintes d'environnement ou de plates-formes d'exécution. Cette exploration est supportée par la technologie synchrone, qui promeut une activité de conception fiable grâce à ses bases mathématiques.

Par la suite, la section 2 introduit d'abord l'environnement GASPARD et le formalisme ARRAY-OL, puis le modèle synchrone flot de données. Ensuite, la section 3 présente les principes de transformation de modèles GASPARD en équations synchrones. Elle est illustrée à l'aide d'un exemple simple dans la section 4. Enfin, la conclusion et les perspectives sont données dans la section 6.

2. Parallélisme de données et approche synchrone

2.1. Les applications de traitement de données intensives et GASPARD

GASPARD (Graphical Array Specification for Parallel and Distributed Computing) [22] est un environnement qui propose une méthodologie de conception conjointe pour les systèmes intégrés sur puce ou SoC, basée sur l'approche MDE (*model-driven engineering*) comme illustré par la Figure 1. Il propose un profil UML intégré au profil MARTE de l'OMG, en cours d'adoption et dédié aux systèmes embarqués et temps réel, qui permet à des utilisateurs de modéliser à la fois des applications de traitement de données intensives et leurs architectures. Un mécanisme d'association est fourni pour les deux aspects, ainsi qu'un ensemble de transformations pour la simulation et la synthèse. Notre approche de modélisation vise à tenir compte de toutes ces caractéristiques de GASPARD et à pouvoir garantir la correction des modèles manipulés au sein de sa méthodologie de conception.

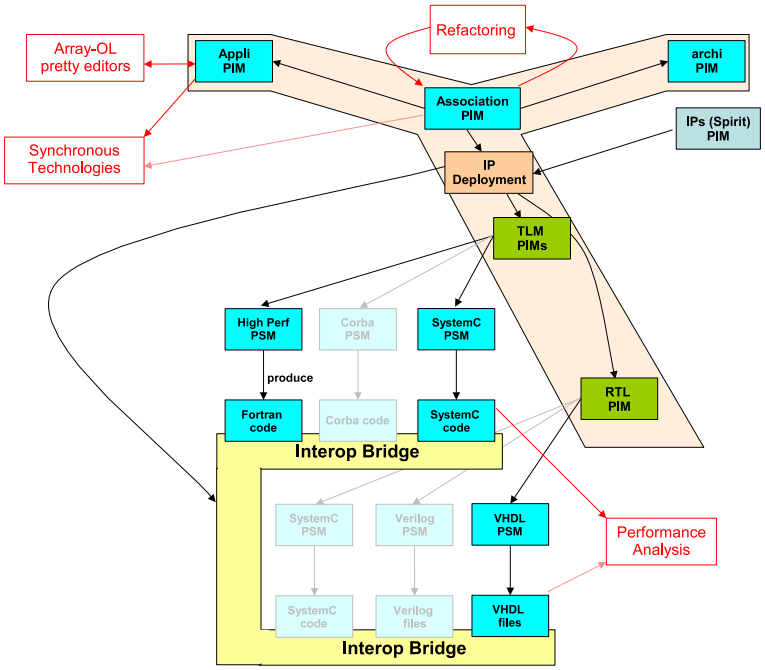


FIG. 1 – Conception de systèmes intégrés sur puce selon GASPARD.

Les modèles GASPARD sont spécifiés à l'aide du langage ARRAY-OL (Array Oriented Language) [8, 21]. Ce dernier permet de modéliser des applications manipulant des quantités importantes de données. Il

adopte des représentations multidimensionnelles permettant d'exprimer tout le parallélisme potentiel présent dans des applications cibles. Les données sont structurées dans des tableaux aux dimensions pouvant être infinies. Une tâche ARRAY-OL consomme et produit des tableaux par morceaux de taille constante, appelés les *motifs*. Des tâches différentes sont reliées entre elles par des dépendances de données. Lorsqu'une dépendance est spécifiée entre deux tâches, cela signifie que l'une d'entre elles requiert des données de la part de l'autre avant de s'exécuter. Ces dépendances exhibent donc initialement un ordre partiel minimal d'exécution. Les applications peuvent être composées hiérarchiquement à différents niveaux de spécifications. En pratique, leurs spécifications consistent en un *modèle global* et en un *modèle local*.

Le modèle global. Le modèle global d'une application est représenté par un graphe acyclique dirigé où les noeuds symbolisent les tâches et les arcs transportent des données à travers des tableaux multidimensionnels. Il n'y a aucune restriction quant au nombre de tableaux d'entrée et de sortie. Les tableaux sont supposés toriques, c'est-à-dire leurs éléments peuvent être consommés ou produits modulo la taille des tableaux. Si le modèle global fournit des informations permettant d'ordonnancer l'exécution des tâches, il n'exprime pas le parallélisme de données présent au sein de ces tâches. Cet dernier aspect est décrit par le modèle local.

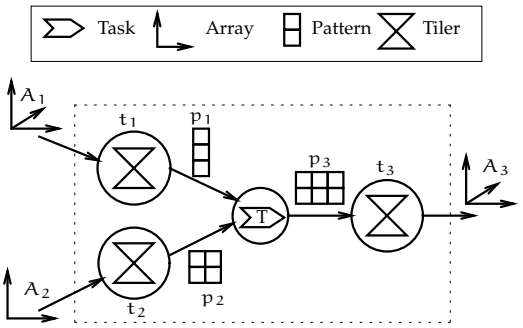
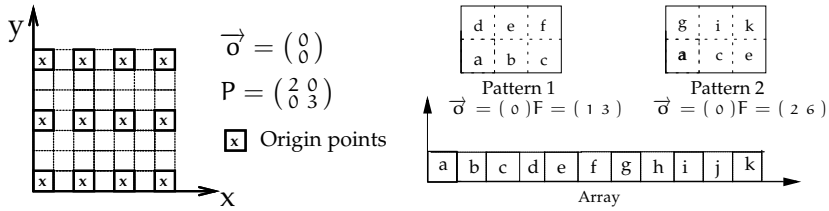


FIG. 2 – Un modèle local en ARRAY-OL.

Le modèle local. Le modèle local décrit le parallélisme de données exprimé à travers des *répétitions*. Une tâche est définie par un constructeur de répétition, où les instances de tâche sont indépendantes les unes des autres. Chaque instance est appliquée à un sous-ensemble d'éléments, appelés motifs ou *patterns*, issus des tableaux d'entrée pour produire des éléments à stocker dans des tableaux de sortie. La façon dont une tâche consomme et produit des tableaux peut être analysée au travers d'un couple (tâche, tableau). De tels couples sont appelés *demi-tâches*. Soit (T, A_i) une demi-tâche, si A_i est un tableau d'entrée, T prend des motifs de A_i pour réaliser son traitement, autrement T stocke ses motifs calculés dans A_i. La taille et la forme des motifs associés à un tableau sont identiques d'une répétition à l'autre. Les motifs peuvent eux-mêmes être des tableaux multidimensionnels. Leur construction est réalisée par l'intermédiaire des *tilers*, qui contiennent les informations suivantes : $\overrightarrow{o}$: origine du motif de référence, $\overrightarrow{d}$: forme du motif (taille de toutes ses dimensions), P : *matrice de pavage* (comment les tableaux sont recouverts à l'aide des motifs), F : *matrice d'ajustage* (comment les motifs sont remplis avec les éléments des tableaux), et $\overrightarrow{m}$: forme des tableaux (taille de toutes ses dimensions).

Chaque couple (tâche, tableau) se voit donc associé à un tiler (voir la figure 2). Pour énumérer les différents modèles, chaque demi-tâche a, par l'intermédiaire de son tiler, une matrice de pavage P et un point de départ représenté par l'origine $\overrightarrow{o}$. La matrice de pavage permet d'identifier l'origine de chaque motif, associée à chaque instance de tâche. Considérons un tableau bidimensionnel avec $\binom{0}{0}$ comme point d'origine et $\left(\begin{smallmatrix}2 & 0\\ 0 & 3\end{smallmatrix}\right)$ comme matrice de pavage. Les constructions de motif commencent alors à partir des positions notées "x" dans la figure 3(a). L'équation 1 exprime le fait que les coordonnées de chaque premier point d'un motif sont calculées comme la somme des coordonnées du point d'origine et d'une combinaison linéaire des vecteurs de pavage, le tout modulo la taille du tableau qui est torique.



(a) Pavage d'un tableau. (b) Exemples d'ajustages.

FIG. 3 – Illustrations du pavage et de l'ajustage.

$$\forall \overrightarrow{x_q},\ \overrightarrow{0} \leq \overrightarrow{x_q} < \overrightarrow{dr}, \overrightarrow{r_q} = (\overrightarrow{o} + P \times \overrightarrow{x_q}) \mod \overrightarrow{m} \quad (1)$$

$\overrightarrow{x_q}$ dénote un motif ayant comme indice q. Nous appelons *espace de répétition*, l'ensemble de tous les indices q possibles lors d'une répétition de tâche. $\overrightarrow{dr}$ contient les bornes de la répétition pour chaque vecteur de la matrice de pavage, c'est-à-dire qu'il délimite l'espace de répétition. Cette information est associée aux tâches. Enfin, $\overrightarrow{r_q}$ représente le point d'origine du motif $\overrightarrow{x_q}$.
La matrice d'ajustage permet, quant à elle, de déterminer des éléments de tableau liés à chaque motif. La figure 3(b) illustre deux exemples d'ajustages en considérant un tableau unidimensionnel, avec (0) comme origine, et où le motif associé est un tableau bidimensionnel de forme $\overrightarrow{d} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$. Un motif peut avoir plus de dimensions qu'un tableau consommé ou produit. Dans le premier cas la matrice d'ajustage est $(1\ 3)$. Chaque vecteur de ce tableau est utilisé pour remplir une dimension du motif. Dans le deuxième cas, la matrice d'ajustage est $(2\ 6)$. Nous observons que les éléments liés à un motif ne sont pas nécessairement consécutifs dans un tableau. Ces éléments sont déterminés par la somme des coordonnées du premier élément du motif et d'une combinaison linéaire de la matrice d'ajustage, le tout modulo la taille du tableau (voir équation 2).

$$\forall \overrightarrow{x_d},\ \overrightarrow{0} \leq \overrightarrow{x_d} < \overrightarrow{d}, (\overrightarrow{r_q} + F \times \overrightarrow{x_d}) \mod \overrightarrow{m} \quad (2)$$

Ici, $\overrightarrow{x_d}$ dénote un élément ayant pour indice d au sein d'un motif.
Comme souligné ci-dessus, les motifs peuvent être eux-mêmes des tableaux multidimensionnels. Cela permet des descriptions hiérarchiques d'applications. Par exemple, la tâche simple T représentée par la figure 2 peut se décomposer en sous-tâches dont les tableaux d'entrées et sorties sont les motifs p_i de celle-ci. Dans la suite, nous utiliserons de façon invariable ARRAY-OL et GASPARD pour qualifier les mêmes modèles.

2.2. Approche synchrone

L'approche synchrone [3] a été proposée dans le but de fournir des concepts formels qui favorisent la conception fiable des systèmes embarqués et temps réel. L'idée de base est que les calculs et communications ayant lieu au sein de différentes réactions d'un système en réponse à son environnement peuvent être considérés comme étant instantanés. Cette idée est communément connue sous l'*hypothèse synchrone*. Ainsi, l'exécution du système est vue à travers la chronologie et la simultanéité de ses événements observés. Cela est différent de la vision de l'exécution d'un système où le temps est considéré sous son aspect chronométrique où la durée joue un rôle significatif.
Le modèle synchrone flot de données. Historiquement, l'origine des langages flot de données peut être associée aux études ayant porté sur les modèles flot de données dans les années soixante dix [9, 13, 24]. C'est le cas des langages synchrones déclaratifs [3] tels que LUSTRE, LUCID SYNCHRONE, ou SIGNAL. Alors que LUSTRE et LUCID SYNCHRONE adoptent un style fonctionnel, SIGNAL est relationnel. Dans ces langages, les données manipulées sont des suites infinies de valeurs. Quelques opérateurs sont fournis pour exprimer des relations entre ces suites. Cela est concrètement représenté sous forme d'équations.

Dans la suite de ce papier, même si la portée de notre étude vise les modèles flot de données synchrones en général, nous considérons principalement LUSTRE [4] pour nos illustrations. Les suites de valeurs

sont manipulées à travers des constructions spécifiques au langage, appelées *nœuds* et identifiées par le mot-clé `node` (voir la section 4 pour une illustration). Le langage LUSTRE étend les fonctions habituelles aux suites, par exemple, addition, multiplication. Des opérateurs spécifiques de manipulation de suites sont également fournis : accès aux valeurs précédentes à partir d'un indice donné, extraction de sous-suites à partir d'une suite, entrelacement de suites, etc.

2.3. Combinaisons de parallélisme intensif et de synchrone

Les premiers travaux sur la représentation de tableaux dans un langage synchrone ont été menés par Halbwachs et Pilaud [12]. Les auteurs utilisent LUSTRE pour concevoir des réseaux systoliques et simuler des algorithmes. Ce travail mena ainsi à l'introduction des tableaux dans LUSTRE car nécessaires pour décrire les algorithmes. Ces tableaux ont été mis en œuvre sur des FPGA par Rocheteau [19]. Plus récemment, Morel proposa une compilation efficace des tableaux en LUSTRE [18]. Notons que si toutes ces études s'intéressent aux tableaux, dans ce papier nous nous intéressons plutôt à la façon dont les langages synchrones flot de données permettent de représenter des applications de traitement de données intensives, manipulant des tableaux multidimensionnels. En particulier, nos représentations en LUSTRE utiliseront les structures de tableaux définis dans le langage.

ALPHA [16] est un langage fonctionnel dédié à la spécifications de calculs réguliers décrits à l'aide de systèmes affines d'équations récurrentes sur des polyèdres. Il permet de raffiner des spécifications ALPHA quelconques de façon à déduire des indices de temps et d'espace pour caractériser une exécution possible. Un environnement est proposé pour la synthèse de circuits à partir de ces spécifications. Les transformations considérées garantissent l'équivalence sémantique entre descriptions initiales et programmes synthétisés, par exemple en C ou en VHDL. ALPHA et ARRAY-OL partagent plusieurs similitudes : les deux langages permettent de spécifier des calculs de données intensives multidimensionnelles tels que les algorithmes numériques ; ils fournissent des environnements de conception proposant des transformations, de la synthèse, etc. Cependant, ils diffèrent sur quelques aspects : ARRAY-OL manipule uniquement des dépendances inter-répétitions uniformes contrairement à ALPHA qui considère également des dépendances non uniformes ; dans ALPHA, l'accès aux données se fait à travers des indices à l'aide de fonctions affines, tandis que dans ARRAY-OL, l'accès est réalisé à travers des motifs, permettant ainsi la hiérarchie et la modularité ; enfin, l'accès cyclique aux données dans les tableaux dans ARRAY-OL est absent dans ALPHA. Dans [20], Smarandache *et al* abordent la validation de systèmes embarqués temps réel en combinant ALPHA et SIGNAL. Dans leur approche, des calculs numériques intensifs sont exprimés à l'aide d'ALPHA. Les contraintes d'horloges résultant des transformations de programmes ALPHA sont spécifiées en SIGNAL. La régularité des descriptions ALPHA permet d'identifier des relations affines entre les horloges. Le compilateur de SIGNAL offre alors la possibilité d'étudier la synchronisabilité des horloges. Dans [11], nous utilisons ces résultats pour analyser les modèles GASPARD. Des concepts similaires à ceux de Smarandache *et al* sont aussi proposés dans [5], où un modèle synchrone est défini pour vérifier la correction-par-construction d'application de traitements de données à hautes performances. Ce modèle permet de synthétiser automatiquement les communications entre des processus aux horloges périodiques non strictement synchrones.

Nous pouvons également citer d'autres langages comme OTTO E MEZZO [17] et STREAMIT [23]. Le premier sert à décrire des comportements de systèmes dynamiques. Il utilise des information d'horloges lors de la génération de code. Le deuxième langage propose une technique efficace de compilation d'applications de traitement de flux de données vers des architectures monoprocessor ou grille de processeur. Ces deux langages qui partagent des points communs avec les langages synchrones, sont aussi dédiés au traitement de données intensives.

Les études précédentes sur la transformation de descriptions ARRAY-OL abordent d'une part des problèmes de compilation pour une génération efficace de code [21], et d'autre part le lien entre le modèle d'ARRAY-OL et celui d'autres formalismes tels que le modèle synchrone multidimensionnel de flux de données de Lee *et al* [10] et les réseaux de processus de Kahn [2]. Enfin, les travaux de Labbani *et al* sur l'introduction du contrôle dans GASPARD [14] peuvent être mentionnés ici. Les auteurs discutent de changements de modes d'exécution dans des applications de traitement de données intensives. La mo-

délisation synchrone de leurs résultats donnera des modèles combinant calculs conditionnés et systèmes de transitions.

3. Des modèles GASPARD vers des équations synchrones

3.1. Schéma général de transformation

La figure 4 résume notre approche. Nous considérons les descriptions exprimées initialement à l'aide du langage de modélisation associé à GASPARD fondé sur le formalisme ARRAY-OL. Il s'agit alors de transformer ces descriptions en systèmes d'équations, traduites dans les langages synchrones flot de données. Ces transformations permettent ainsi d'accéder aux fonctionnalités offertes à la fois par les environnements associés à GASPARD (descriptions logicielles et matérielles, transformations, différents types de simulation...) et aux langages synchrones (analyse et vérification formelles, etc).

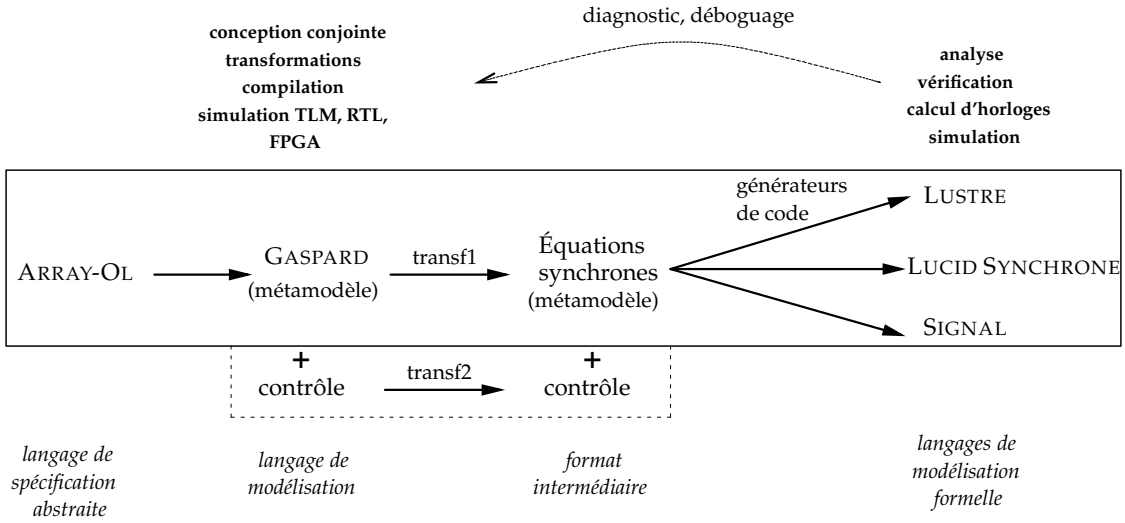


FIG. 4 – Schéma général de transformation.

Dans ce papier, nous nous concentrons particulièrement sur la transformation *transf1* de la branche mise en évidence par le rectangle représenté sur la figure 4. Elle permet d'associer une représentation sous forme d'équations synchrones à des modèles d'applications à parallélisme de données exprimées en ARRAY-OL. Elle est décrite sous forme de règles de transformations conformément au principe du MDE adopté dans l'environnement GASPARD. Les modèles synchrones qui en résultent, ainsi que leur principe d'obtention sont exposés dans la section 3.2. La transformation *transf2*, qui n'est pas présentée ici, sera construite sur les mêmes bases que *transf1*, en prenant en compte les aspects liés au contrôle dans les applications à parallélisme de données en s'inspirant de travaux antérieurs dédiés à l'introduction de mécanismes de contrôle dans GASPARD [14].

3.2. Modèle en équations synchrones

Les modèles synchrones flot de données sont très proches des modèles GASPARD du point de vue structurel. Par conséquent, modéliser les seconds à l'aide des premiers, en utilisant les équations et la composition synchrone devient simple.

3.2.1. Représentation des tableaux ARRAY-OL

Les tableaux d'ARRAY-OL sont modélisés par des flots de données de type tableau. En effet, une particularité de ces tableaux est qu'ils peuvent avoir une dimension infinie, et aucune représentation explicite du temps n'est donnée. En fait, cette dimension infinie des tableaux est souvent utilisée pour décrire un flot infini de tableaux constitués des dimensions restantes. Ainsi, nous considérons concrètement cette représentation pour obtenir un flot de tableaux.

3.2.2. Représentation d'une répétition dans le domaine de répétition

Pour une tâche transformant des données reçues à partir de deux tableaux A_1 et A_2, en données dans un tableau A_3 (voir figure 2), la répétition peut être décrite par :

$$A_3[< ind_3^j >] = T(A_1[< ind_1^j >], A_2[< ind_2^j >]) \quad (3)$$

où ind_i^j dénote l'indice correspondant à un point j dans l'espace de répétition ; et T est le modèle synchrone de la partie calcul, c'est-à-dire la tâche, qui peut être vue comme un appel à une fonction externe dans un langage synchrone. Nous supposons que les flots de tableaux synchrones considérés ici permettent des affectations par élément. La modélisation d'un calcul répété revient alors à la composition synchrone des équations associées à chaque point de l'espace de répétition.

Une autre particularité d'ARRAY-OL est qu'il ne représente que les dépendances de données, laissant par conséquent disponible tout le parallélisme potentiel dans les spécifications. Plus particulièrement, les répétitions appliquées aux tableaux décrivent combien de fois, et selon quels pavages et ajustages un calcul doit être effectué. Par contre, elle n'imposent aucun ordre quant à la façon d'accéder aux éléments des tableaux manipulés. En d'autres termes, *n'importe quel ordre* respectant les dépendances de données pourrait convenir dans le cas d'une exécution séquentielle. Nous nous conformons à cette propriété d'ARRAY-OL en utilisant simplement la composition synchrone des modèles des différentes répétitions associées à une tâche, tout en n'induisant aucun ordre quelconque entre celles-ci.

Il est possible d'avoir une représentation plus compacte de ce modèle, en utilisant des opérateurs spécifiques aux tableaux disponibles dans les langages équationnels synchrones considérés. Par exemple, l'application de l'opérateur *map* à la fonction T offre une notation plus compacte [18]. Le modèle que nous présentons ici est censé être simple. Celui-ci est certainement optimisable. Cependant, il est important de noter que les optimisations des équations synchrones peuvent varier suivant la nature de l'étude qu'on souhaite faire par la suite : model-checking, génération de code, etc.

3.2.3. Restructuration du modèle parallèle simple

Le modèle décrit ci-dessus peut être restructuré, de façon à avoir une meilleure correspondance avec la structure intrinsèque d'une tâche répétitive ARRAY-OL, composée de tilers d'entrée, de calculs, et de tilers de sortie. Un avantage d'une telle restructuration est l'obtention d'une transformation structurelle d'ARRAY-OL en équations synchrones, qui est facilement implémentable. Un autre avantage est lié à l'introduction du contrôle, qui est l'une des perspectives à court terme de notre étude, en s'inspirant des résultats préliminaire de [14]. En effet, si l'on considère le contrôle d'une tâche répétée à l'aide d'automates dont les états correspondent à une configuration ou un mode, et les transitions entre ces modes, il est alors possible d'envisager le contrôle différemment pour chacun des tilers, mais aussi pour la partie calcul de la tâche.

Par ailleurs, cette restructuration repose sur les propriétés de l'opération de composition synchrone qui est commutative et associative. De plus, les signaux intermédiaires ajoutés par la restructuration n'induisent aucun coût supplémentaire car il peuvent être détectés et isolés facilement lorsqu'ils sont inutiles grâce aux optimisations des compilateurs synchrones. L'équation 3 donnée ci-dessus peut être ainsi être séparée en trois parties, correspondant respectivement :

- aux tilers d'entrée, décrivant les équations qui réalisent l'extraction des motifs d'entrée pour chaque point de l'espace de répétition : $p_1^j = A_1[< ind_1^j >]$ et $p_2^j = A_2[< ind_2^j >]$;
- au calcul, qui produit les motifs de sortie pour chaque point de l'espace de répétition : $p_3^j := T(p_1^j, p_2^j)$;
- aux tilers de sortie, décrivant les équations qui réalisent l'insertion des motifs de sortie pour chaque point de l'espace de répétition : $A_3[< ind_3^j >] = p_3^j$.

Au final, nous obtenons le modèle illustré par la figure 5. Grâce à la modularité des langages synchrones, notamment par les propriétés de l'opération de composition qu'ils offrent, nous pouvons décrire des modèles d'applications plus complexes, en composant en parallèle ou hiérarchiquement, des modèles de tâches tel que celui illustré sur la figure 5. Dans les deux cas, le parallélisme inhérent au modèle de

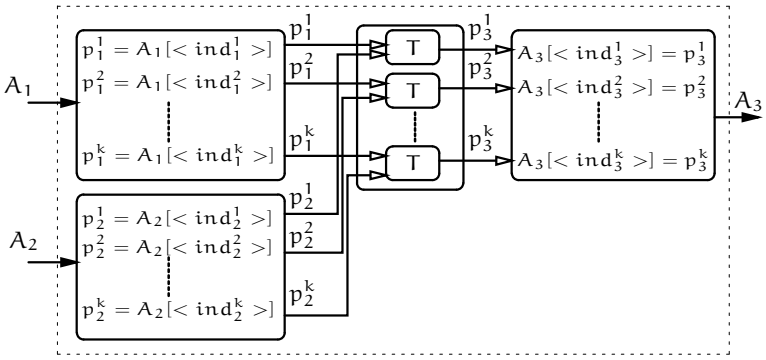


FIG. 5 – Modélisation des tilers et des tâches parallèles.

spécification ARRAY-OL/GASPARD reste préservé.

4. Illustration sur un exemple

4.1. Présentation informelle

L'exemple considéré concerne le traitement d'images. Le premier traitement effectué sur chaque image, appelé `Phase1` sur la figure 6, consiste en deux opérations : 1) une *réflexion horizontale* ou miroir horizontal, suivie d'une *rotation à 90° dans le sens horaire*. Le second traitement (ou `Phase2`), appliqué à l'image est identique au précédent. Il a pour but de renvoyer l'aspect originel de l'image.

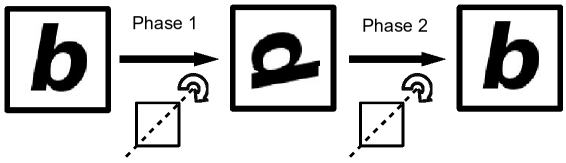


FIG. 6 – Illustration informelle de l'exemple.

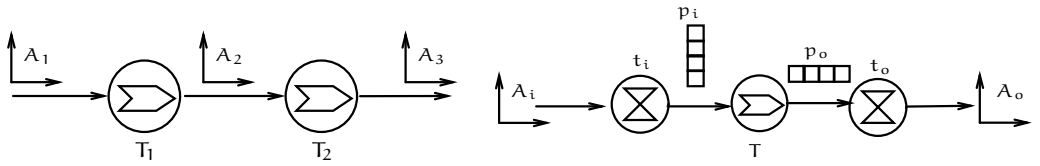
Dans cet exemple, les images sont traitées suivant les colonnes. Chaque colonne est considérée comme un motif dans la spécification GASPARD. Ainsi, le traitement associé à chaque phase peut être exprimé de la façon suivante : d'abord, l'image est consommée suivant ses colonnes qui représentent les motifs ; ensuite, pour chaque motif en entrée, un motif est calculé en sortie ; enfin, l'image est reconstruite à partir des motifs calculés.

Pour des raisons de simplicité, nous considérons une matrice de taille 4*4 pixels pour illustrer le modèle synchrone résultants.

4.2. Les modèles GASPARD de l'exemple

Cet exemple est modélisé en GASPARD dans la figure 7(a). Dans le modèle global, nous distinguons deux tâches T_1 et T_2. Celles-ci représentent des instances d'une même tâche. Les entrées et les sorties pour T_1 et T_2 sont également de même nature. Par conséquent, dans ce qui suit, nous ne nous concentrons que sur la tâche T_1.

Le modèle local correspondant à chacune des tâches T_1 et T_2 est illustré sur la figure 7(b) - par exemple, pour T_1, A_i et A_o sont respectivement représentés par A_1 et A_2). Les informations associées au modèle GASPARD sont les suivantes :



(a) Modèle global de l'exemple. (b) Modèle local d'une tâche de l'exemple.

FIG. 7 – Modèles GASPARD/ARRAY-OL de l'exemple.

- tableau d'entrée A_i : [4,4]
- tiler d'entrée t_i : $\overrightarrow{o}$:[0,0] ; $\overrightarrow{d}$: [4] ; P : [1,0] ; F : [0,1] ; $\overrightarrow{m}$: [4,4]
- la tâche T : fonction qui ne transfère que les données sans les calculer. Et son domaine de répétition, $\overrightarrow{dr}$: [4]
- tiler de sortie t_o : $\overrightarrow{o}$:[0,0] ; $\overrightarrow{d}$: [4] ; P : [0,1] ; F : [1,0] ; $\overrightarrow{m}$: [4,4]
- tableau de sortie A_o : [4,4]

La transformation de cet exemple en équations synchrones suivant le principe exposé dans la section 3.1 est présentée dans la section suivante. Nous indiquons le code généré associé dans le langage LUSTRE.

4.3. Résultat de la transformation en LUSTRE

Nous commençons par définir des types associés aux différents tableaux manipulés dans l'exemple. A priori, ces tableaux n'ont pas forcément les mêmes dimensions. En LUSTRE, un type tableau, par exemple `TA1`, est défini par : `type TA1 = int^A1_TD[0]^A1_TD[1]`. Celui-ci est associé à A_1. Il a deux dimensions ayant pour tailles `A1_TD[0]` et `A1_TD[1]`. `A1_TD` est la taille des dimensions des tableaux ; et ses éléments sont de type entier (`int`). Les définitions de type de motifs, par exemple : `type TMOTIF1 = int^tiler1_mo_td;` où `tiler1_mo_td` est la taille du motif, sont identiques à celles du type des tableaux. Le nombre de répétitions de la tâche T_1 est donné par la constante `task1_nr`, déclarée comme suit : `const task1_nr = TACHE1_DR;` où les valeurs `TACHE1_DR` dénotent les dimensions de l'espace de répétition, c'est-à-dire, les composantes de $\overrightarrow{dr}$.

Le code LUSTRE donné sur la figure 8 illustre le modèle synchrone généré après transformation du modèle GASPARD. Seul le corps principal du code est donné. `node GLOBAL_MODEL` représente le modèle global qui contient deux modèles locaux `LOCAL_MODEL_LMi`$_{\{i=1,2\}}$, et leurs dépendances de données sont représentées par `A1, A2` et `A3`. Dans le `LOCAL_MODEL_LM1`, les motifs `M1, M2, M3` et `M4` sont construits par un nœud `LM1_TILER_IN` qui remplit la fonction de tiler d'entrée. La tâche `LM1_TASK` consomme les motifs précédents et produit les motifs de sortie `M5, M6, M7` et `M8`. Le tableau de sortie `AO` est construit par `LM1_TILER_OUT`. Les constructions de motifs dans `LM1_TILER_IN`, et de tableaux de sorties à partir des motifs de sorties suivent le principe expliqué et illustré en figure 5.

Enfin, le nœud `EXTERNAL_PROC_LM1` effectue un appel à une fonction externe réalisant un traitement sur des motifs. Dans notre exemple, cette fonction consiste simplement en un transfert de données entre motifs d'entrée et de sortie.

5. Discussion

L'approche présentée dans cet article montre nos premiers résultats sur la modélisation et la validation d'applications de traitement de données intensives en utilisant l'approche synchrone. Ces résultats sont prometteurs. Ils fournissent une base intéressante permettant de lier deux modèles de spécification : d'une part ARRAY-OL, un langage dédié à l'expression de parallélisme de donnée, et les langages synchrones flot de données. Ces modèles offrent des concepts puissants pour aborder la conception et la validation d'applications massivement parallèles. L'environnement GASPARD, qui utilise ARRAY-OL comme formalisme de description, permet de modéliser des systèmes embarqués à travers leurs

aspects fonctionnels et leurs architectures. Quant aux langages synchrones, ils favorisent la validation formelle pour une conception sûre. Notre approche diffère des travaux de Smarandache *et al* [20] par le fait que nous proposons un modèle synchrone d'applications à parallélisme de données, incluant aspects fonctionnels et non fonctionnels, alors que dans [20], les auteurs se focalisent principalement sur la question des horloges, donc des aspects non fonctionnels. Par ailleurs, nos modèles sont spécifiés entièrement et de façon exclusive dans chacun des langages synchrones LUSTRE, LUCID SYNCHRONE et SIGNAL. Cela permet ainsi d'accéder aux techniques et outils associés à chacun de ces langages. Dans le cas de [20], les auteurs combinent SIGNAL et ALPHA, n'accédant par conséquent qu'aux outils de SIGNAL réservés à l'analyse d'horloges. Toutefois, nous empruntons certains de leurs résultats dans notre approche [11].

```
--Modele global
node GLOBAL_MODEL(A1: TA1)
 returns(A3: TA3); var A2: TA2;
let
  A2 = LOCAL_MODEL_LM1(A1);
  A3 = LOCAL_MODEL_LM2(A2);
tel

--Modele local
node LOCAL_MODEL_LM1(AI: TA1)
 returns(AO: TA2);
 var M1, M2, M3, M4: TMOTIF1;
     M5, M6, M7, M8: TMOTIF2;
let
  (M1,M2,M3,M4)=LM1_TILER_IN(AI);
  (M5,M6,M7,M8)=LM1_TASK(M1,M2,M3,M4);
  AO = LM1_TILER_OUT(M5,M6,M7,M8);
tel

--Tiler d'entree
node LM1_TILER_IN(AI: TA1)
 returns(AO1,AO2,AO3,AO4: TMOTIF1);
let
  AO1 = PROC_TILER1_0(AI);
  AO2 = PROC_TILER1_1(AI);
  AO3 = PROC_TILER1_2(AI);
  AO4 = PROC_TILER1_3(AI);
tel

--Tache T1
node LM1_TASK(AI1,AI2,AI3,AI4: TMOTIF1)
 returns(AO1,AO2,AO3,AO4: TMOTIF2);
let
  AO1 = EXTERNAL_PROC_LM1(AI1);
  AO2 = EXTERNAL_PROC_LM1(AI2);
  AO3 = EXTERNAL_PROC_LM1(AI3);
  AO4 = EXTERNAL_PROC_LM1(AI4);
tel

--Tiler de sortie
node LM1_TILER_OUT(M1,M2,M3,M4 :
 TMOTIF2) returns(AO: TA2);
let
  AO = PROC_TILER2(M1,M2,M3,M4);
tel

--Construction d'un motif
node PROC_TILER1_0(AI: TA1)
 returns(AO: TMOTIF1);
 --ici, les indices ont été calculés
 --lors de la transformation
let
  AO[0] = AI[0,0];
  AO[1] = AI[0,1];
  AO[2] = AI[0,2];
  AO[3] = AI[0,3];
tel

node PROC_TILER1_1(AI: TA1)
 returns(AO: TMOTIF1);
node PROC_TILER1_2(AI: TA1)
 returns(AO: TMOTIF1);
node PROC_TILER1_3(AI: TA1)
 returns(AO: TMOTIF1);
...

--Construction de tableau de sortie
node PROC_TILER2(AI0,AI1,AI2,AI3: TMOTIF2)
 returns(AO: TA2);
let
  AO[0,0] = AI0[0];
  AO[1,0] = AI0[1];
  AO[2,0] = AI0[2];
  AO[3,0] = AI0[3];
  AO[0,1] = AI1[0];
    ...
tel

--Appel externe effectue par la tache
node EXTERNAL_PROC_LM1(AI: TMOTIF1)
 returns(AO: TMOTIF2);
```

FIG. 8 – Code LUSTRE associé à l'exemple.

La seconde étude proche de la nôtre est celle de Cohen *et al* [5]. Ici, les auteurs proposent une notion particulière d'horloge pour la description du contrôle dans les applications de traitement de données intensives. Ils définissent des horloges dites *n-synchrones*, modélisant des calculs périodiques dans des applications de traitements à hautes performances. Cela offre ainsi la possibilité d'aborder des questions de synchronisabilité dans des contextes où le synchrone classique ne permet d'apporter de réponse. Notre approche vise également à intégrer ces résultats dans nos modèles.

Enfin, bien que les approches d'OTTO E MEZZO [17] et de STREAMIT [23] s'intéressent aux applications de traitement de données intensives comme la nôtre, elles ont des objectifs différents. Dans la première approche, il s'agit de spécifier et de simuler des systèmes dynamiques, tandis que dans la seconde, les auteurs se concentrent sur des problèmes de compilation efficace d'applications de traitement de flot de données à hautes performances.

Parmi les questions importantes auxquelles nous comptons nous confronter, il y a l'introduction de concepts adéquats permettant de manipuler explicitement le contrôle dans les applications à parallélisme de données. Le contrôle sera introduit premièrement sous la forme de dépendances conditionnées, où une dépendance de données sera valide lorsque sa condition associée est vraie. Il aura donc une sémantique basée sur l'opérateur `when` de SIGNAL ou LUSTRE. Avec ces formes de contrôle dans les spécifications GASPARD, les techniques synchrones telles que le calcul d'horloges seraient applicables pour l'analyse et la compilation, avec la gestion d'horloges multiples, typiquement pour la génération du code distribué. Ensuite, ce contrôle pourra être décrit sous la forme de modes de calculs dans lesquels différentes façons seront proposées pour effectuer des calculs sur des entrées pour produire des sorties. Ces modes pourront se différencier suivant différents critères, par exemple la façon dont les données sont accédées dans les tilers. En se basant sur le travail préliminaire de Labbani *et al* [14], nous pourrons spécifier des automates hiérarchiques et parallèles en définissant des modes et des transitions entre ces modes. Les systèmes de transitions résultants peuvent être analysés en utilisant les techniques de vérification tels que le model-checking. D'autre part, ce travail pourra mener à l'application de techniques de synthèse de contrôleur discret. En effet, grâce à une introduction propre des structures de contrôle des tâches [1, 7], cette technique permettra de générer automatiquement une partie du gestionnaire de tâches pour renforcer des propriétés de sûreté dans le système.

6. Conclusions et perspectives

Dans cette étude, nous proposons des transformations en équations synchrones d'applications à parallélisme de données au sein de l'environnement GASPARD, dédié à la conception conjointe de systèmes intégrés sur puces. Ces transformations suivront une approche d'Ingénierie Dirigée par les Modèles (IDM/MDE). Elles fournissent ainsi des modèles synchrones sur lesquels des raisonnements formels peuvent être menés en utilisant l'arsenal d'outils et de techniques disponibles dont dispose la technologie synchrone pour la validation, simulation, génération de test, distribution etc.

À propos du contrôle, la technique de représentation basée sur les automates proposés dans le cadre des languages synchrones flot de données est intéressante. Par exemple, dans [6], les automates sont introduits de façon hiérarchique, avec un style à la automates de mode [15]. Cela rend possible l'expression de mécanismes complexes de contrôle lors de la conception. Ceux-ci feront alors, via les modèles synchrones, un plein usage des outils que sont le calcul d'horloge, le model-checking, ou la synthèse de contrôleurs discrets.

Bibliographie

1. Karine Altisen, Aurélie Clodic, Florence Maraninchi, et Éric Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the European Symposium on Programming, ESOP'03,* April 7 - 11, 2003,Warsaw, Poland, number 2618 in LNCS, pages 174–188. Springer Verlag, 2003.
2. A. Amar, P. Boulet, et P. Dumont. Projection of the Array-OL specification language onto the Kahn

process network computation model. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, Las Vegas, Nevada, USA*, December 2005.
3. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, et R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1) :64–83, January 2003.
4. P. Caspi, D. Pilaud, N. Halbwachs, et J.A. Plaice. LUSTRE : a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'87)*. ACM Press, 1987.
5. A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, et M. Pouzet. N-sychronous Kahn networks. In *ACM Symp. on Principles of Programming Languages (PoPL'06)*, Charleston, South Carolina, USA, January 2006.
6. Jean-Louis Colaço, Bruno Pagano, et Marc Pouzet. A Conservative Extension of Synchronous Dataflow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.
7. Gwenael Delaval et Eric Rutten. A domain-specific language for multi-task systems, applying discrete controller synthesis. In *Proceedings of the 21st ACM Symposium on Applied Computing, SAC 2006* Dijon, France, April 23-27 2006, 2006.
8. A. Demeure et Y. Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis conference on Micro-Electronics (SAME'98), System-on-Chip Session, France*, October 1998.
9. J.B. Dennis. First version of a data flow procedure language. In *Programming Symposium, LNCS 19, Springer Verlag*, pages 362–376, 1974.
10. P. Dumont et P. Boulet. Another multidimensional synchronous dataflow : Simulating ARRAY-OL in PTOLEMY II. Technical Report RR-5516, INRIA, France, March 2005. `www.inria.fr/rrrt/rr-5516.html`.
11. A. Gamatié, E. Rutten, H. Yu, P. Boulet, et J.-L. Dekeyser. Synchronous modeling of data intensive applications. Technical Report RR-5876, INRIA, France, April 2006. `www.inria.fr/rrrt/rr-5876.html`.
12. N. Halbwachs et D. Pilaud. Use of a real-time declarative language for systolic array design and simulation. In *International Workshop on Systolic Arrays*, Oxford, July 1986.
13. G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress, vol 74 of Information Processing*, pages 471–475, 1974.
14. O. Labbani, J.-L. Dekeyser, P. Boulet, et E. Rutten. Introducing control in the Gaspard2 data-parallel metamodel : Synchronous approach. In *Int'l Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES'05), Montego Bay, Jamaica*, October 2005.
15. F. Maraninchi et Y. Rémond. Mode-automata : a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, (46) :219–254, 2003.
16. C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes I, France, December 1989.
17. O. Michel, D. De Vito, et J.-P. Sansonnet. 8 1/2 : data-parallelism and data-flow. In *Proc. of the 9th International Symposium on Lucid and Intensional Programming*, 1996.
18. L. Morel. Efficient compilation of array iterators in LUSTRE. In *First workshop on Synchronous Languages, Applications and Programming*, Grenoble - France, April 2002.
19. F. Rocheteau et N. Halbwachs. POLLUX, a LUSTRE-based hardware design environment. In P. Quinton et Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, June 1991.
20. I.M. Smarandache, T. Gautier, et P. Le Guernic. Validation of mixed SIGNAL-ALPHA real-time systems through affine calculus on clock synchronisation constraints. In *World Congress on Formal Methods (2)*, pages 1364–1383, 1999.
21. J. Soula, P. Marquet, J.-L. Dekeyser, et A. Demeure. Compilation principle of a specification language dedicated to signal processing. In *6th International Conference on Parallel Computing Technologies (PaCT'2001)*, Novosibirsk, Russia, September 2001. LNCS 2127, Springer Verlag.
22. The WEST Team. `www.lifl.fr/west/gaspard`.
23. W. Thies, M. Karczmarek, et S. Amarasinghe. StreamIt : A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, 2002.
24. W.W. Wadge et E.A. Ashcroft. LUCID*, the dataflow programming language.* Academic Press, 1985.