INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Another Multidimensional Synchronous Dataflow: Simulating Array-OL in Ptolemy II

Philippe Dumont and Pierre Boulet
LIFL, USTL
Cité Scientifique
59 655 Villeneuve d'Ascq Cedex

Email: Philippe.Dumont@lifl.fr, Pierre.Boulet@lifl.fr

N° 5516

March 1, 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5516--FR+ENG

Another Multidimensional Synchronous Dataflow: Simulating Array-OL in Ptolemy II

Philippe Dumont and Pierre Boulet
LIFL, USTL
Cité Scientifique
59 655 Villeneuve d'Ascq Cedex
Email: Philippe.Dumont@lifl.fr, Pierre.Boulet@lifl.fr

Thème COM — Systèmes communicants
Projet DaRT

Rapport de recherche n° 5516 — March 1, 2005 — 19 pages

Abstract: Computation intensive multidimensional applications appear in many application domains such as video processing or detection systems. We present here the Array-OL specification model to handle such multidimensional applications. This model is compared to the Multidimensional Synchronous Dataflow proposition by Lee et al.

We also detail in this a new domain in the Ptolemy simulation environment dedicated to Array-OL specification simulation.

Key-words: Multidimensional Dataflow, Ptolemy, signal processing, computation model

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 72 92 59 ??

Array-OL : un nouveau langage pour les flots de données synchrones multidimensionnels

Résumé : Nous introduisons dans cet article le model de spécification Array-OL qui permet de gérer des applications à flots de données multidimensionels pour le traitement du signal. Nous comparons également Array-OL à GMDSDF le seul modèle équivalent dans ce domaine. De plus nous proposons un nouveau « domaine » Ptolemy dédié à la simulation d'applications décrites en Array-OL.

Mots-clés : Flots de données multidimensionnels, Ptolemy, traitement du signal, modèle de calcul

1 Introduction

Computation intensive multidimensional applications are predominant in several application domains such as image and video processing or detection systems (radar, sonar). In general, many intensive signal processing applications are multidimensional. By multidimensional, we mean that they manipulate primarily multidimensional data structures such as arrays. For example, a video is a 3D object with two spatial dimensions and one temporal dimension. In a sonar application, one dimension is the temporal sampling of the echoes, another is the enumeration of the hydrophones around a submarine and others such as frequency dimensions appear during the computation. Actually, such an application manipulates a stream of 3D arrays.

Dealing with such applications presents a number of difficulties:

- Very few computation models are multidimensional;
- The access patterns to the arrays are diverse and complex;
- Scheduling such an application with bounded resources and time is challenging, especially in a distributed context.

As far as we know, only two computation models have attempted to propose formalisms to model and schedule such multidimensional signal processing applications: MDSDF [6, 2, 11, 12] and Array-OL [4, 3]. MDSDF and its follow-up GMDSDF have been proposed by Lee and Murthy. They are extensions of the SDF model proposed by Lee and Messerschmitt [8, 9]. Array-OL has been introduced by Thomson Marconi Sonar and its compilation has been studied by Demeure, Soula, Dumont et al. [14, 13, 5]. Array-OL is a specification language allowing to express all the parallelism of a multidimensional application, included the data parallelism, in order to allow an efficient distributed scheduling of this application on a parallel architecture.

The goals of these two propositions are similar and though they are very different on their form, they share a number of principles such as:

- Data structures should make the multiple dimensions visible;
- Static scheduling should be possible with bounded resources;
- The application domain is the same: intensive multidimensional signal processing applications.

In section 2 we will introduce the SDF based models: SDF, MDSDF and GMDSDF. In section 3 we will describe Array-OL: how it works and how it tries to solve the drawbacks of the previous models. In section 4 we compare both models. We then explain how we have implemented a simulation environment for Array-OL in Ptolemy [7] in section 5. Finally, we illustrate our implementation on an example in section 6.

2 SDF and its multidimensional extensions

In this section we will explain the main ideas behind the already existing SDF model. We will see the principles of the 1-D SDF model and we will see its limitations. Then we will study the multidimensional SDF model.

2.1 Synchronous Dataflow

The synchronous dataflow model (SDF) has been created and developed by Edward A. Lee since 1986. Lee has integrated it in his modeling and simulation environment for embedded system: Ptolemy.

SDF allows to model simple dataflow systems. In SDF, an application can be described as an acyclic oriented graph, each node consumes the data on its incoming edges and produces data on its

outputs edges. In Ptolemy the nodes are called "actors" and represent operations that are applied on the data carried on the edges. These data elements are called "tokens".

The main characteristics of an SDF application are:

- An actor always consumes and produces the same amount of tokens at each execution. This amount must be known at modeling time. But no information is needed on the operations made by the actors.
- The values of the data must not influence the flow of data. These values are just used by the actors for the computations but they can't change the behavior of the application. In other words, there is no data-dependent control flow.

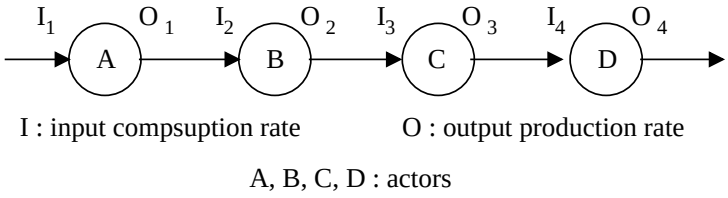

Figure 1: A simple SDF application

Hence the application is staticly defined. Using this property we can take advantage of the formal properties of the model. It is possible:

- to schedule the applications at modeling time,
- to have a determinate execution,
- to detect deadlocks,
- to execute the application in a finite time and to use a finite amount of memory.

Computation of the scheduling: We call I_x and O_x the number of tokens consumed and produced by an execution of the x-th task and we define r_x as the number of executions (Fig.1). We can write the following equation (called the *balance equation*) for each task:

$$r_x O_x = r_{x+1} I_{x+1}$$

This system of equations has no solution or an infinity. In the first case the SDF graph has inconsistent rates and in the second case all the solutions are multiple from the smallest one. It is possible to write these solutions (k is called the *blocking factor*) like:

$$k\overrightarrow{r} = k \begin{pmatrix} r_1 \\ r_2 \\ \vdots \end{pmatrix}$$

So with one of these solution we will be able to do a good scheduling of an SDF application. Using the results of [8] a precedence graph can be constructed automatically.

A good example is shown in Fig.2. The first task produces two tokens while the second task consumes three tokens. The smallest solution of the balance equation is $\overrightarrow{r} = \binom{3}{2}$ and the dependence graph obtained is shown in the figure.

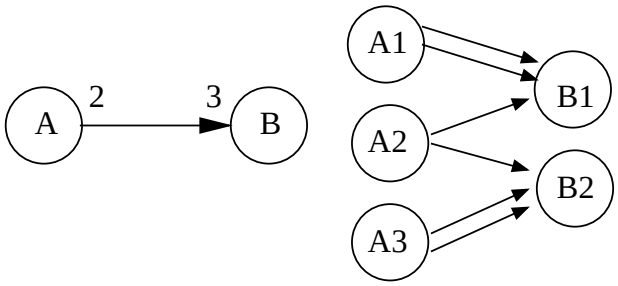

Figure 2: 1-D SDF and is precedence graph

Utilization of state and delay: Another possibility of the SDF model is the utilization of delays. A delay is a set of initial tokens on an edge. The production of tokens by the task located just before it on the edge will be offset, but the consumption of the task located just after it will not change. A delay of two means that the first and the second execution of the consumption task will take the initial values present on the edge and will take, afterwards, the result of the production task.

A state is a self loop on a task which allows to use the result of an iteration in the next one. Of course for the first iterations there must be a default value like with delays.

States and delays do not change the computation of the scheduling.

2.2 MDSDF: Multidimensional Synchronous Dataflow

The SDF level allows us to model 1-D streams by specifying the dependences between the actors. The tokens carried by these streams can be simple values but they can also be vectors or arrays. The principle of MDSDF is not to carry multidimensional tokens because it is already possible with SDF, but to express the number of tokens produced and consumed in a stream with more than one dimension.

The principles of MDSDF are quite similar to those of SDF. On each task we just have to specify the number of data consumed and produced on each dimension. Fig.3 illustrates with a simple example the manipulation of MDSDF.

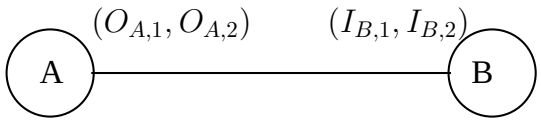

Figure 3: A MDSDF graph

The use of MDSDF allows us to express all the meaning of our applications. Of course it is possible to still model some applications (Fig.4) with the SDF model (Fig.5) but the description is more complicated to understand.

There is also more hidden problems. Fig.6 shows a simple application which can not be modeled in 1-D SDF even with a linearization of the dimensions. The reason is very simple, it's not possible in SDF to specify how to build the precedence graph.

Computation of the scheduling: The system of equations of the SDF model is replaced by several systems of equations. The equations for Fig.3 are:

$$r_{A,1}O_{A,1} = r_{B,1}I_{B,1}$$
$$r_{A,2}O_{A,2} = r_{B,2}I_{B,2}$$

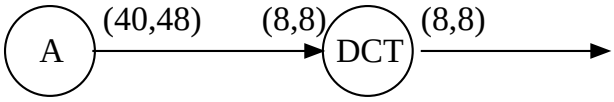

Figure 4: An image processing application in MD-SDF.

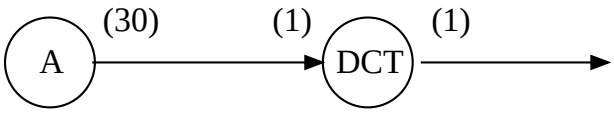

Figure 5: The same image processing application in SDF.

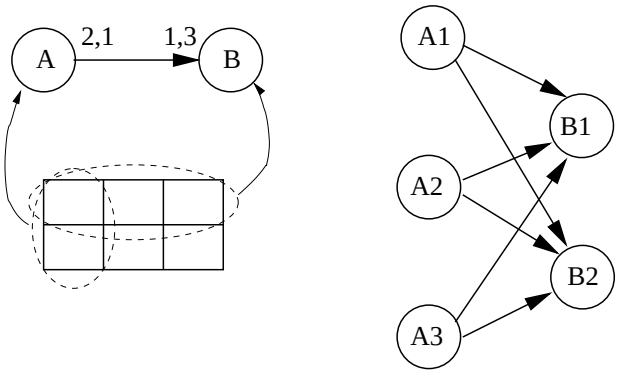

Figure 6: Precedence

These systems must be solved independently. The solutions obtained have the same form as the solutions for the 1-D SDF model. The first system gives the number of iterations for the first dimension and the second equation for the second dimension. The total number of iterations of a task is equal to the product of the number of iterations for each dimension.

Particularity of the MDSDF model: For the moment the number of dimensions is always the same in the whole application there is no way to specify the creation or the suppression of dimension. That's why Lee has introduced several "key actors". No operations are done during the execution of these key actors, they are just here to control how the data structures are made. For example the "downsample" allows to discard a dimension (Fig.7)

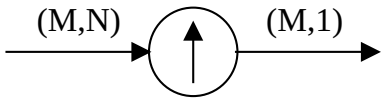

Figure 7: Downsample

Like in SDF, it is possible to use states and delays in MDSDF. But now the states and delays are multidimensional tuples and represent initial rows and columns. There is no other differences and their manipulation is still the same.

Conclusion: The MDSDF model allows to use multidimensional streams, but there are limitations on these streams! The consumption and the production of data must be parallel to the axes. In an attempt to remove this restriction Praveen K. Murthy and Edward A. Lee have proposed an extension to MDSDF called GMDSDF.

2.3 GMDSDF: Generalized Multidimensional Synchronous Dataflow

With MDSDF the consumption or the production of data were necessarily parallel to the axes. The goal of GMDSDF is to provide tools to model non parallel to the axes consumption or production. The idea of GMDSDF is to produce points on a lattice. The form of such a lattice is decided by special task called source and can only be modified by two specials tasks called the decimator and the expander. We will shortly give some definitions just before explaining these three special tasks and we will finally show how to compute the scheduling.

2.3.1 Definitions:

***LAT*(V): the lattice generated by the sampling matrix V:** Given a square matrix V called the sampling matrix, the lattice $LAT(V)$ generated by V is the set of points $\overrightarrow{t} = V.\overrightarrow{n}, \forall \overrightarrow{n} \in \mathbb{N}^m, m \in \mathbb{N}$.We can consider that in a 2-D MDSDF model, $L = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$.

***FDP*(V): the fundamental parallelepiped:** Given a sampling matrix, the fundamental parallelepiped is the parallelogram drawn by the column vectors of V sketched from the origin. $FDP(V)$ is the set of points $V.\overrightarrow{x}$ where $\forall i \in \mathbb{N},\ 0 \leq i < m,\ 0 \leq x_i < 1,\ x_i \in \mathbb{R}$.

N(V): is the set of integers points inside $FDP(V)$.

The renumbered points of *LAT*(V): Given a point n on $LAT(V)$, there exits an integer vector $\overrightarrow{k}$ such that $\overrightarrow{n} = V.\overrightarrow{k}$. The set of points described by $\overrightarrow{k}$ is called the renumbered points of $LAT(V)$. Fig.8 shows the renumbered points and their corresponding points on the lattice.

The support matrix: In order to describe very quickly the renumbered points, we introduce the notion of support matrix. Given a sampling matrix V and a matrix Q as support matrix, the renumbered points are the set $N(Q)$.

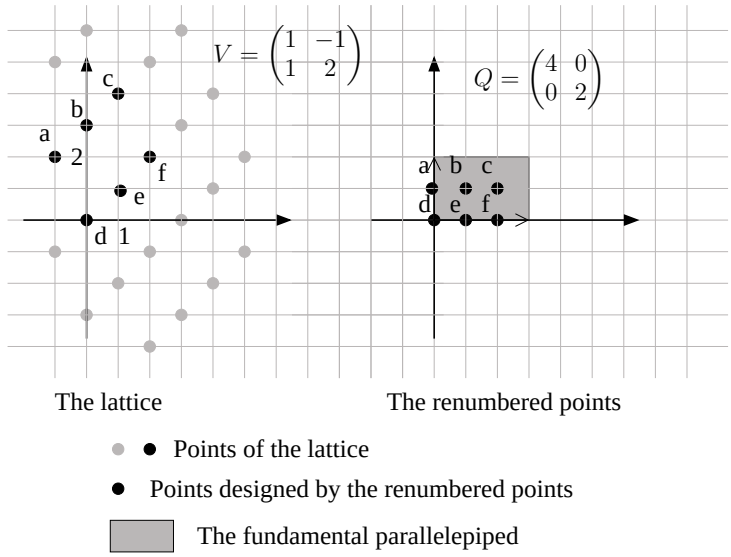

Figure 8: Most important definitions to understand the GMDSDF model

The containability condition: Given X a set of integer points in $\mathbb{R}^m$, X satifies the containability condition if there exists an $m * m$ rational-valued matrix W such that $N(W) = X$.

2.3.2 Source, decimator and expander:

Using all these definitions, we will now introduce the three special tasks used in GMDSDF.

The source: it defines the form of the lattice using a sampling matrix V and produces data on this lattice at each firing. The set of renumbered points matching the data produced has to satisfy the containability condition. In other words, for each source there is a support matrix Q such that the points of $N(Q)$ are the renumbered points of the source. So a source is fully defined with its sampling lattice and its support matrix.

Decimator and expander: The source allows to create a lattice, the decimator and the expander allows to modify the form of this lattice. The decimator drops points and the expander adds points.

We note V_e and V_f respectivly the incoming and the outgoing sampling matrices. In the same way, we note W_e and W_f the support matrices. These matrices are used to define the decimator and the expander but the decimator also needs a decimation matrix M and the expander an expansion matrix L.

The relationships between the input and output lattices is given by

Decimator $V_f = V_e.M \quad W_f = M^{-1}.W_e$

Expander $V_f = V_e.L^{-1} \quad W_f = L.W_e$

Using the previous equations, it is possible to deduce the outgoing matrices using the incoming ones. Moreover in a GMDSDF application the lattice can only be changed by a source, a decimator or an expander. Hence it is possible to compute the incoming sampling matrix (the input lattice) by following the manipulation done on the lattice from the source which generated it.

So the decimators and the expanders can be simply defined with their decimation or expansion matrix and their input support matrix.

2.3.3 States and delays:

A state or a delay is interpreted as an offset in the renumbered data space. The fundamental parallelogram needed to compute the renumbered points is not based at the origin but is offset by the vector given with the state or the delay.

2.3.4 Computation of the scheduling:

In MDSDF it was possible to compute simply the scheduling by solving a system of equations. This system was simple to write because for each actor we knew the number of samples consumed and produced on each dimension and these dimensions were parallel to the axes. But due to non parallel consumption and production, it is more complicated in GMDSDF.

Simplification of the specification model: In order to be able to compute the scheduling in GMDSDF, we will do a lot of simplifications. All the simplifications describe below are only for 2-D applications, Lee never gives examples for applications with more dimensions.

We consider that each actor produces or consumes a generalized (x, y) rectangle of data. The data are taken on the lattice and are placed in this rectangle according to a rectangularizing function.

The expander is simplified: it consumes a $(1, 1)$ rectangle and produces a $(L1, L2)$ rectangle where $L1$ and $L2$ are two positive integers such that $L1L2 = |det(L)|$.

The decimator is also modified it consumes a $(M1, M2)$ rectangle, which is defined according to the previous actors in the task, and produces a $(1, 1)$ rectangle in average (due to the form of the lattice, there is not always a point to keep in the rectangle).

Computation of the scheduling:

GMDSDF is an extension of MDSDF and it seems that the more convenient way to compute the scheduling is to consider the GMDSDF applications as those of MDSDF. The generalized rectangles are used to write a system of equations like in MDSDF. But this not enough, others constraints must be resolved for the decimators and for the directions of the repetitions of each actor. These problems are too complicated to be explained in this paper.

The dependence graphs are constructed after the resolution of all these constraints.

Modeling the applications:

So we can summarize the modeling of a GMDSDF application as follow:

- Creation of the acyclic graph and placement of the actors
- Choice of the sampling matrix on the sources and choice of the generalized rectangle
- Computation of the repetition number of each actor.
- Computation of the support matrices.

The support matrices are computed after the scheduling because they are written as functions of the lattice and of the repetition number of an actor. So the applications are fully defined after the computation of the scheduling.

2.3.5 Conclusion

As Edward Lee said in his paper: the definition of a GMDSDF will not be easy to use in a programming environment. One of the problems is that only the expander and the decimator can change the lattices. So the modifications of the lattices and the computations on the data of these lattices are done by actors which are at the same level of modeling. The consumption of data on these lattices is very regular and does not allow to take care of more complicated patterns of data. We introduce below the multidimensional modeling model called Array-OL as an alternative to GMDSDF.

3 Array-OL Model of Specification

In this section we will study the modeling language Array-OL. It's important to notice that Array-OL is only a specification language, no rules are specified for executing an application described with Array-OL. But a scheduling can be easily computed using this description.

The basic principles that underly the language are:

- Array-OL is a data dependence expression language.
- All the available parallelism in the application should be available in the specification, both task parallelism and data parallelism.
- It is a single assignment formalism. No data element is ever written twice. It can be read several times, though.
- The spatial and temporal dimensions are treated equally in the arrays. In particular, time is expanded as a dimension (or several) of the arrays.
- The arrays are seen has tori. Indeed, some spatial dimensions may represent some physical tori (think about some hydrophones around a submarine) and the frequency domains obtained by FFT are toroidal.

The modeling of an application in Array-OL needs two levels of description. The first one is the global model, it defines the task parallelism in the form of dependences between tasks and arrays. The second one is the local model which details the elementary action the tasks realize on array elements. This local model expresses the data parallelism.

3.1 Global model

The global model (Fig.9) is a simple directed acyclic graph. Each node represents a task and each edge an array. The number of incoming or outgoing arrays is not limited. Moreover the number of dimensions of these arrays is not related between the inputs and the outputs. So a task can consume two two-dimensional arrays and produce a three-dimensional one. The creation of dimensions by a task is very useful, a very simple exemple is the FFT which creates a frequency dimension.

There is only one limitation on the dimensions: there must be only one infinite dimension by array. Most of the time, this infinite dimension is used to represent the time, so having only one is quite sufficient.

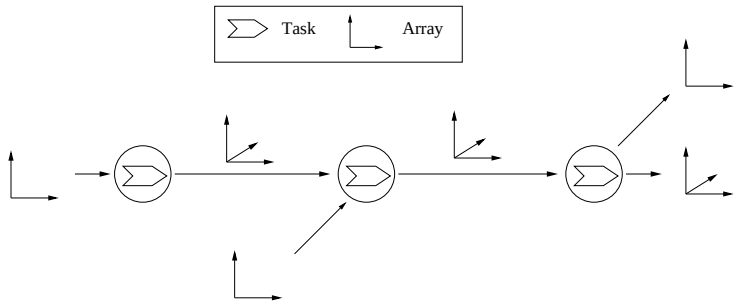

Figure 9: A global model

At the execution of a task, the incoming arrays are consumed and the output arrays are produced. But the number of arrays produced or consumed is equal to one for each edge. It's not possible to

consume more than one array for producing one. The graph is a dependence graph, not a data flow graph.

So it is possible to schedule the execution of the tasks just with the global model. But it's not possible to express the data parallelism of our applications because the details of the computation realized by a task are hidden at this specification level.

3.2 Local model

The local model is a little bit more complicated. At this level, we specify how the data are consumed and produced in a task. This process is not realized in one pass. In fact the data elements are treated block by block.

So to execute a local model there are several repetitions which can be described as follows:

- a block is extracted from each input array (the size of the blocks can be different for each array)
- the computation is made with these blocks
- each resulting block is stored in its destination array (there must be a resulting block for each destination array).

As we can see, for each repetition, a block is extracted from each input array and a result block is stored in each output array. The size and shape of a block associated to an array is the same from an repetition to another. That's why we called a block of data a *pattern*. In order to all a hierarchical construction, the patterns are themselves arrays.

In order to give all the information needed to create these patterns, we have associated to each array (ie each edge) a tiler. A tiler is able to build a pattern from an array, or to store a pattern in an array. It contains the following informations:

- $\overrightarrow{o}$: the origin of the reference pattern (for the reference repetition)
- $\overrightarrow{d}$: the shape (size of all the dimensions) of the pattern
- P: a "paving" matrix
- F: a "fitting" matrix
- $\overrightarrow{m}$: the shape (size of all the dimensions) of the array

Now with all these informations we are able to do all the manipulations around our notion of pattern.

3.2.1 How to fill a pattern?

From a reference element ($\overrightarrow{r}$) in the array, one can extract a pattern by enumerating its other elements relatively to this reference element. We will use the fitting matrix to compute the others elements. The coordinates of the elements of the pattern are build as the sum of the coordinates of the reference element added to a linear combination of the fitting vectors.

$$\forall \overrightarrow{x_d},\ \overrightarrow{0} \leq \overrightarrow{x_d} < \overrightarrow{d}\,,\ \overrightarrow{r} + F \times \overrightarrow{x_d}$$

Vector $\overrightarrow{d}$ gives the bounds of the linear factors on each dimension of the fitting matrix. Fig.11 gives several examples of fitting matrices and patterns.

A key element one has to remember when using Array-OL is that all the dimensions of the arrays are toroidal. That means that all coordinates of pattern points are computed modulo the size of the array dimensions.

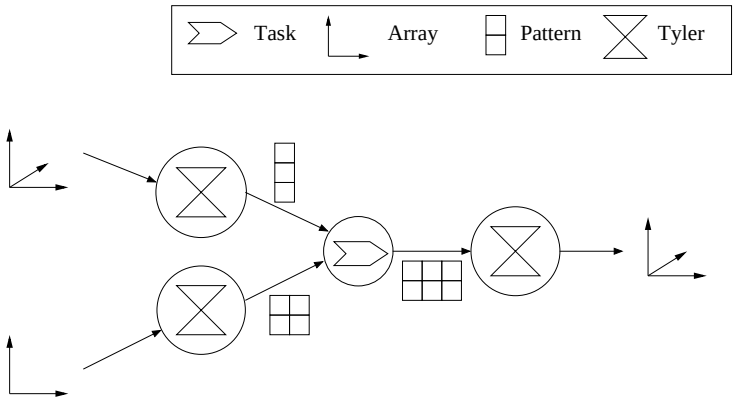

Figure 10: A local model

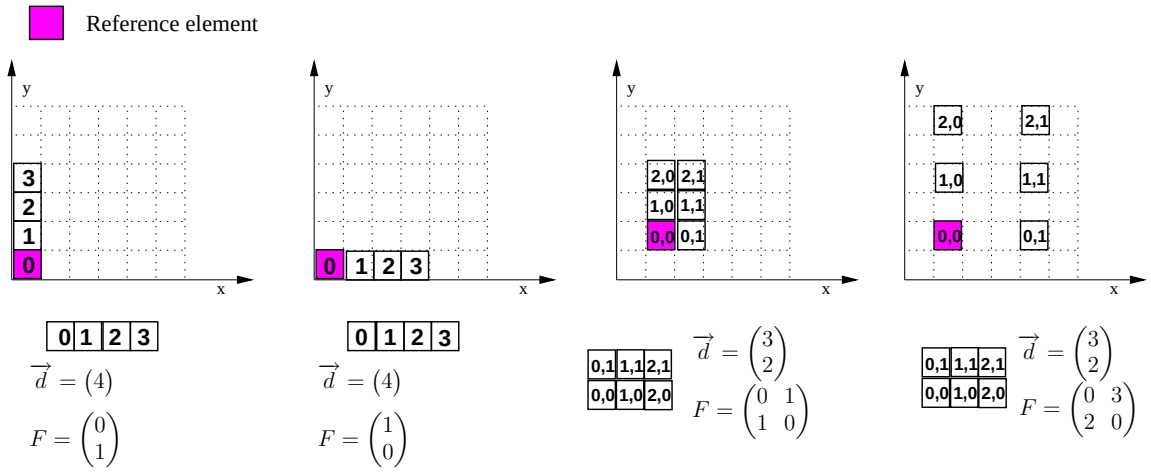

Figure 11: Construction of a pattern

3.2.2 Paving an array with patterns

For each repetition, one needs to design the reference elements of the input and output patterns. We use for that a similar scheme as the one used to enumerate the elements of a pattern.

The reference elements of the reference repetition are given by the $\overrightarrow{o}$ vector of each tiler. The other reference elements are build relatively to this one. As above, their coordinates are built as a linear combination of the vectors of the paving matrix.

Fig.12 shows some examples. On this figure the reference elements of the pattern have a number to distinguish them. But this number is not an order of execution. It is important to understand that each repetition is independent from the others. Hence it is possible to distribute each repetition for each task of our global model and to compute them in parallel.

3.2.3 How many repetitions?

But there is still one question: how many repetitions is there in a local model? The answer is quite simple: just enough to fill an output array. In fact in Array-OL, all the elements of an output array must be computed exactly once! So if we want to know the number of repetitions we just have to use the paving matrix and the size of the array to deduce vector $\overrightarrow{q}$ which contains the bounds of repetition for each vector of the paving matrix.

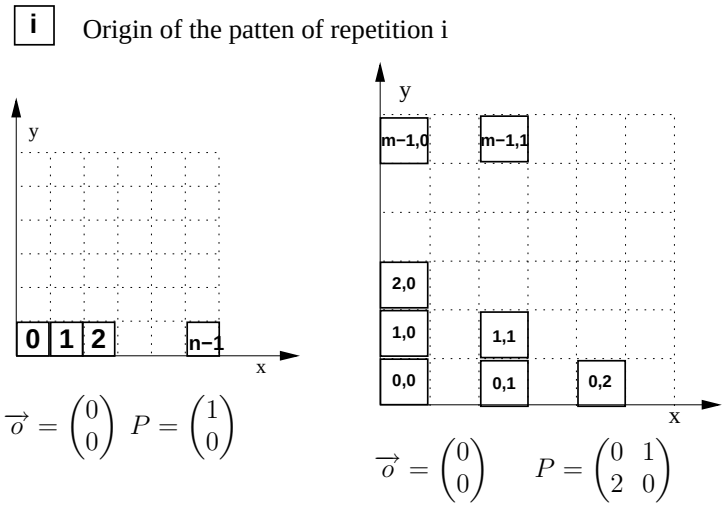

Figure 12: Origin points of a pattern

3.2.4 Summary and non trivial exemples

We can summarize all these explanations with two formulas:

- $\forall \overrightarrow{x_q}, \overrightarrow{0} \leq \overrightarrow{x_q} < \overrightarrow{q}, (O + P \times \overrightarrow{x_q}) \mod \overrightarrow{m}$ give all the reference elements of the patterns.
- $\forall \overrightarrow{x_d}, \overrightarrow{0} \leq \overrightarrow{x_d} < D, (O + P \times \overrightarrow{x_q} + F \times \overrightarrow{x_d}) \mod \overrightarrow{m}$ enumerates all the elements of a pattern for the $\overrightarrow{x_q}$ repetition.

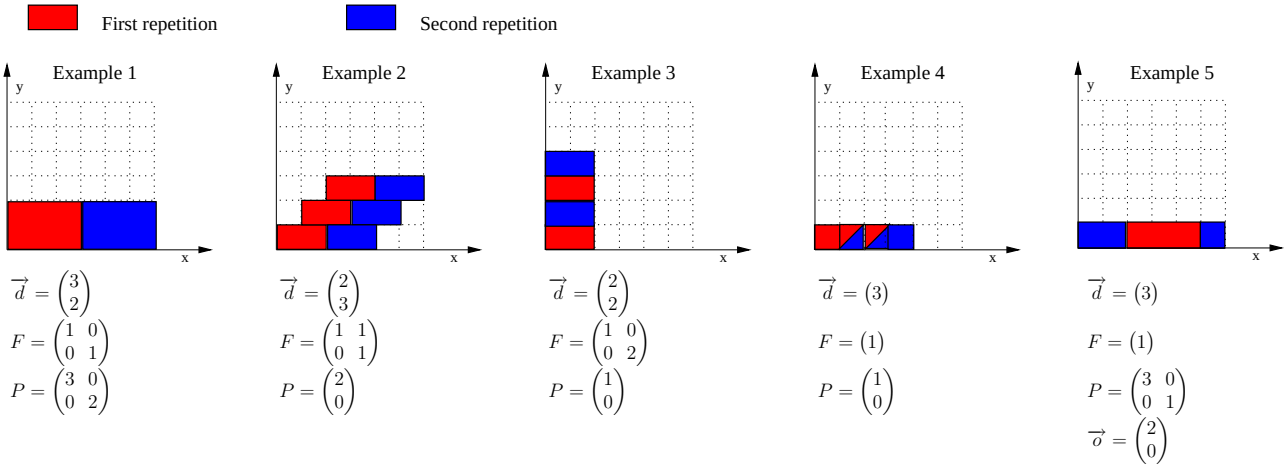

Figure 13: Different repetitions

In order to show the very interesting possibilities of the local model, we will illustrate it with a few exemples: all the examples of Fig.13 show only two repetitions.

1. A very simple example.
2. The points of a pattern are not necessarily parallel to the axes.
3. The points of a pattern can be discontiguous.

4. The points of two different patterns are not automaticly distinct.
5. The array are toroidal and the origin here is not $\begin{pmatrix}0\\0\end{pmatrix}$ but $\begin{pmatrix}2\\0\end{pmatrix}$.

3.3 Hierarchy

According to the specification of the local model, Array-OL applications must respect some obligation on the pattern shape: it's not possible to create non parallelepipedic patterns. But it's possible to compute a bouding box around such patterns. This allows to build an application in a hierarchical way. Indeed the computation made in a local model can be done not only by a simple task like a fast fourier transformation but by a whole Array-OL task with a global and a local model.The incoming and outgoing arrays of the hierarchical tasks are the patterns of the up tasks.

The data dependences visible at a given hierarchical level are approximations of the real data dependences. One needs to look at the whole hierarchical construction to have the precise dependences. One striking example is a global model with one task linking two infinite arrays. The dependences expressed here are that one array depends on the other. The see the dependences between the elements, one has to look into the local model. There the dependences are expressed between patterns (or sub-arrays). If the repetitive task is hierarchical, one would need to go down the hierarchy to precise these data dependences.

3.4 Summary

Array-OL allows us to describe systematic signal processing applications in a very convenient way. More precisely we describe dependences. At the global level, the dependences between tasks are given by the input and output arrays. At the local level, the dependences are given in terms of patterns. Once again, Array-OL is a specification model and do not impose any execution order.

4 Comparison of GMDSDF and Array-OL

The comparison of GMDSDF and Array-OL must be done in two parts. On the one hand we have to consider which one is the most expressive and convenient to describe multidimensional applications. On the other hand, we need to compare the ways to computate the scheduling of these applications.

4.1 Descriptions of applications

In Array-OL the points are stored into or taken from arrays using patterns. These patterns must be themselves arrays, that means that they must be parallelepipedic (Fig.12) which is not always the case in GMDSDF. We are not sure that this feature of GMDSDF is really useful and we can argue that it is always possible to take a bouding box in such situations. On the other hand, it is possible in Array-OL to have non contiguous pattern (Fig.12, example 3) which is not possible in GMDSDF.

Another possibility of Array-OL which is not present in GMDSDF is the utilization of toroidal arrays (Fig.12, example 5). It is a very useful feature that we have already used for different applications.

However, with GMDSDF it is possible to use states and delays. Array-OL really lacks that feature. We are actually working on an extension which will support this feature.

A description in Array-OL has another advantage: the manipulations on the arrays and the computations on data are clearly distinguished. In GMDSDF it is possible to find a decimator followed by an FFT at the same level of specification. On the other hand in Array-OL all the treatments on arrays are handled by the tilers.

We can also say that an application is easier to describe in Array-OL, but we are not sure to be very objective.

4.2 Computation of the scheduling

In GMDSDF, the computation of the scheduling is really challenging and the specification of an application is ended only when the scheduling has been computed (due to the support matrix). In Array-OL all the application can be modeled without thinking about its scheduling. Moreover in Array-OL it is possible to take benefit of the simple execution order present in each application. The only thing to do is to execute the local models one by one (see section 5 for an implementation), and a local model can also be distributed to exploit the data parallelism expression present in the model. We propose such a distributed execution scheme in [1].

4.3 Conclusion

Both models have strengths and weaknesses and none includes the other. The Array-OL model is easier to understand and allows to separate the specification of the application and its scheduling.

5 Implementation in Ptolemy

Ptolemy has been developed in the University of Berkley since 1993. It provides an environment to model and simulate applications for embedded systems. It's possible to do the simulation with different computation models, these models are called domains and one of them is the SDF model. There is no implementation of the MDSDF and GMDSDF models.

Our idea is to simulate Array-OL in Ptolemy in order to have a simple but efficient simulation tool. In a first time we have to decide how to represent the different element of an Array-OL specification: the global and the local models, the arrays, the tilers, ... In a second time we have to decide which domains will be used to direct the simulations.

The representation of the global model (Fig.14) is very simple, we represent each task by a composite actor. Such actors are just empty boxes with input and output ports, they are used for creating hierarchical applications. Inside these composite actors, we place the local models. We do not show the arrays because they are only tokens and not actors.

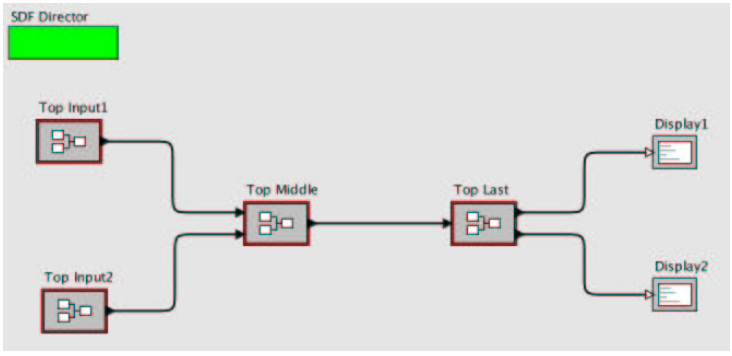

Figure 14: A global model representation in Ptolemy

The representation of the local model (Fig.15) is not more complicated. The tilers are represented by actors which can be configured with a menu (Fig.16). The task itself is also represented by an actor. It is possible to replace this actor by a composite one in order to create a hierarchy.

The tilers and the tasks are both shown as actors in spite of their differences of semantic. Indeed, we were obliged to represent the tilers by actors because they are doing a computation and so they need to be executed. We have also introduced tiler histograms to display how many iterations of paving have been done on a tiler.

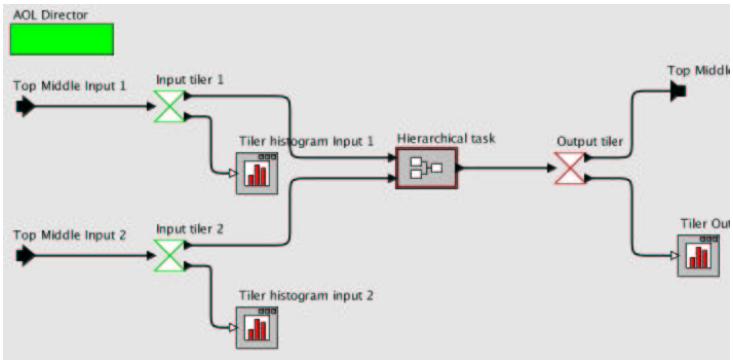

Figure 15: A local model representation in Ptolemy

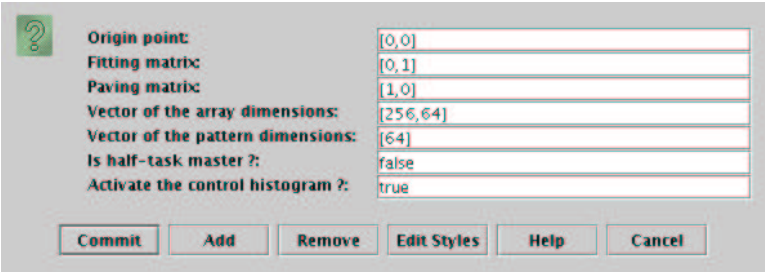

Figure 16: Configuration of a tiler

Ptolemy allows to use different model of execution: Synchronous DataFlow (SDF), Process Network (PN), Discrete Event (DE), ... These models are called domains. To execute an application with such a model we only have to select a dedicated actor called "a director". SDF is clearly the most appropriate model. In SDF, a global model is easy to simulate with a consumption and a production rate equal to one. The main thing to do is only to linearize the arrays.

Our first idea was to do the same thing for the local model. But in a SDF domain an actor can be executed only one time and after it has to wait for the others to be executed. It is a problem because when all the actors of a local model has been executed one time, the SDF director of the global model launches the execution of the next local model instead of waiting for the execution of the first one to be finished. So we have created an Array-OL director which is an extension of a SDF director and which is able to execute correctly a local model.

A first version, based on the version 3.0.2 of Ptolemy, has been released in September 2004. All the features of Array-OL are not supported like the arrays with infinite dimensions, and the optimization of the applications is not available in this version. But we hope that a new version will be finished for June 2005.

6 Example

In this section we study Array-OL on a concrete application. This application has been used by Lee in [11] to illustrate the utilization of GMDSDF. Originally, this example was drawn from [10].

As we are not very experienced with such applications, we hope that we have understood it correctly. Any suggestions will be greatly appreciated if we have done some mistakes.

6.0.1 Introduction

The goal of this application is to convert the format of a $2:1$ interlaced TV signals from a $4/3$ aspect ratio to a $16/9$ aspect ratio. The relationship between the picture height P_h and the picture width P_w is $P_h = \frac{3}{4}P_w$ with a $4/3$ aspect ratio and $\overline{P_h} = \frac{9}{16}P_w$ with a $16/9$ aspect ratio. The number of lines ($N = 650$) is the same in the two cases so we can established the following relationship for the interline distances d_y:

$$\overline{d_y} = \frac{\overline{P_h}}{N} = \frac{3}{4}\frac{P_h}{N} = \frac{3}{4}d_y$$

So a possibility to do the conversion between the two aspect ratios is to map $\frac{3}{4} \times N$ lines of the input images, with $4/3$ aspect ratio, into N lines on the output images with $16/9$ aspect ratio. Of course there are different ways to do this but we will study here the multidimensional approach proposed in [10].

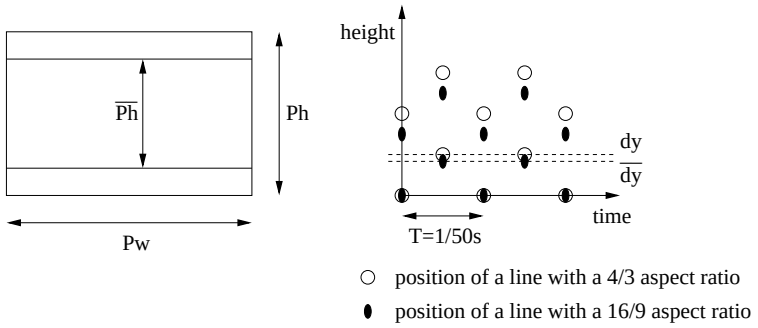

Figure 17: Representation of the two formats and of the verticotemporal plane

A video signal can be seen as a three-dimensional signal. These three dimensions are the height, the width and the time. The current applications usualy sample the signal on two of these three dimensions and let the others contiguous. Most of the time the horizontal dimension is kept contiguous while the verticotemporal plane, composed by the vertical and the time dimensions, is sampled.Hence a dot in the verticotemporal plane is a line on the TV screen.

In [10] the authors propose to proceed the conversion in three steps : two interpolations followed by a simple decimation (Fig.18).The two interpolations are used to generate new intermediate lines according to the input ones. The decimation only select the good lines to generate the image on the output screen.

The parameters of the two interpolations are given in the application description: the first one requires 11×5 lines to produce 4 lines and the second one needs 3 lines to produce 2 lines.Finally, the decimation only keeps one line on six with a shift of three lines on the time dimension for creating the scanline effect. Using this informations, we have decided to model this applications with Array-OL.

6.1 Description in Array-OL

Array-OL is able to deal only with arrays, so we have to decide how to represent the application data. In fact it is very simple, the TV signal can already be considered as an array with three dimensions. But the signal is interlaced, so our array will have empty spaces (Fig.18). In real condition it is not

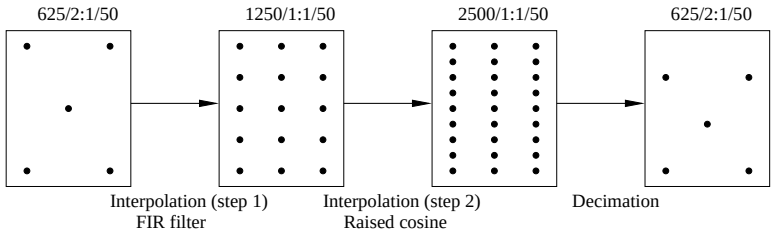

Figure 18: Description of the application using schemas of the verticotemporal planes

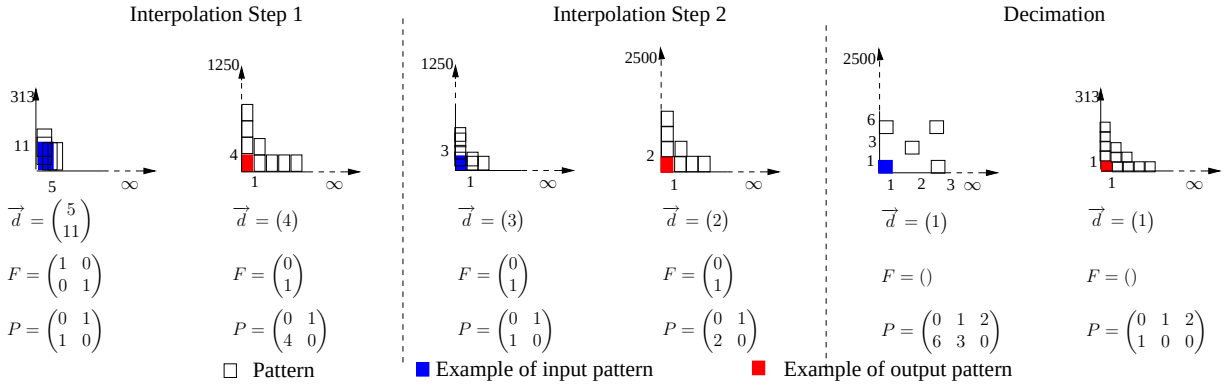

Figure 19: The application in Array-OL

the case, so we consider that the height of the array is equal to $\frac{625}{2}$ and that there is no empty spaces.

As the global one is evident, we will just explain here the different local models. The first and the second interpolations are very easy to model (Fig 19) because there is just to apply the informations given in [10] (see section 6.0.1). The decimation is more complicated to model. The difficulty is due to the shift of the data consumption between two columns of the verticotemporal plane. The solution is to use three vectors of paving. The first vector is used to take the points on the height dimension. The second vector expresses the shift and is used only once[1]. The third vector allowes to move from the first column to the third one, then to the fifth one, and so on.

We will not study here the simulation of this application, it will be done in future work.

As we can see, it is quite simple to describe an application in Array-OL. Array-OL allow to stay very close to the application definition.

7 Conclusion

We have presented in this paper two multidimensional specification models: GMDSDF and Array-OL. Both models allow to describe intensive signal processing applications such as video processing or detection systems. We have detailed both models and their mathematical notations and compared them.The presentation has been illustrated by the description of the implementation of a simulation domain for Array-OL in Ptolemy and an example.

Future works about Array-OL will deal with adding states and delays to the model and studying the impact of this addition on the code transformation techniques used to optimize the scheduling of Array-OL applications.

[1]The bounds of iteration for the paving dimensions are not explained in this paper.

References

[1] A. Amar, P. Boulet, and P. Dumont. Projection of the Array-OL specification language onto the Kahn process network computation model. Extended version `http://www.lifl.fr/west/publi/ABD05i.pdf`, 2005.

[2] M. J. Chen and E. A. Lee. Design and implementation of a multidimensional synchronous dataflow environment. In *1995 Proc. IEEE Asilomar Conf. on Signal, Systems, and Computers*, 1995.

[3] A. Demeure and Y. Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis conference on Micro-Electronics (SAME 98)*, France, Oct. 1998.

[4] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J.-C. Dufourd, and J.-L. Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, Sept. 1995.

[5] P. Dumont and P. Boulet. Transformations de code Array-OL : implémentation de la fusion de deux tâches. Technical report, Laboratoire d'Informatique fondamentale de Lille et Thales Communications, Oct. 2003.

[6] E. 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, Jan. 1993. North-Holland.

[7] E. A. Lee. *Overview of the Ptolemy Project*. University of California, Berkeley, Mar. 2001.

[8] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, Jan. 1987.

[9] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proc. of the IEEE*, Sept. 1987.

[10] R. Manduchi, G. M.Cortelazzo, and G. A. Mian. Multistage sampling structure conversion of video signals. *IEEE Transactions on circuits and systems for video technology*, 1993.

[11] P. K. Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. PhD thesis, University of California, Berkeley, CA, 1996.

[12] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, July 2002.

[13] J. Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, Dec. 2001. (In French).

[14] J. Soula, P. Marquet, J.-L. Dekeyser, and A. Demeure. Compilation principle of a specification language dedicated to signal processing. In *Sixth International Conference on Parallel Computing Technologies, PaCT 2001*, pages 358–370, Novosibirsk, Russia, Sept. 2001. Lecture Notes in Computer Science vol. 2127.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399