INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Projection of the Array-OL Specification Language onto the Kahn Process Network Computation Model

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

Email: Abdelkader.Amar@lifl.fr, Philippe.Dumont@lifl.fr,

Pierre.Boulet@lifl.fr

N° 5515

Mars 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5515--FR+ENG

Projection of the Array-OL Specification Language onto the Kahn Process Network Computation Model

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

Thème COM — Systèmes communicants
Projet DaRT

Rapport de recherche n° 5515 — Mars 2005 — 18 pages

Abstract: The Array-OL specification model has been introduced to model systematic signal processing applications. This model is multidimensional and allows to express the full potential parallelism of an application: both task and data parallelism. The Array-OL language is an expression of data-dependences and thus allows many execution orders.

In order to execute Array-OL applications on distributed architectures, we show here how to project such specification onto the Kahn process network model of computation. We show how Array-OL code transformations allow to choose a projection adapted to the target architecture.

An experiment on a distributed process network implementation based on CORBA concludes this article.

Key-words: Multidimensional Dataflow, signal processing, computation model, Kahn process network, CORBA

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 ??

Projection du langage de spécification Array-OL sur les réseaux de processus de Kahn

Résumé : Le modèle de spécification Array-OL a été créé pour décrire des applications de traitement du signal systématique. Il s'agit d'un modèle multidimensionnel permettant d'exprimer le parallélisme d'une application, que se soit le data parallélisme ou le parallélisme de tâche. De plus, Array-OL étant un langage d'expression de dépendances, il est possible d'avoir plusieurs ordres d'exécution.

Afin de pouvoir exécuter Array-OL sur des architectures distribuées, nous proposons ici une projection d'Array-OL sur les réseaux de processus de Kahn en utilisant ces derniers comme modèles de calcul. Nous introduisons également des transformations qui permettent d'optimiser cette projection en fonction de l'architecture cible.

Nous concluons en donnant un exemple basé sur une implémentation CORBA des réseaux de processus.

Mots-clés : Flots de données multidimensionnels, traitement du signal, modèle de calcul, réseaux de processus de Kahn, CORBA

1 Introduction

Signal processing dedicated to detection systems refers to multidimensional arrays. As in digital sound processing, a first dimension allows to sample the signal in chronological order. A second dimension generally represents the different sensors, the temporal sampling is applied on each of them. During the signal processing, others dimensions may appear. For example, during the FFT implementation a new dimension represents the frequency. The temporal reference is modified and matches the sampling of the different FFT execution ages. Designing an optimized distributed implementation of such multidimensional applications is very challenging.

In order to conform to the needs for specification, standardization and efficiency of the multidimensional signal processing, Thomson Marconi Sonar has developed a signal processing oriented language: Array-OL (Array Oriented Language) [10, 9]. This application domain is characterized by systematic, regular, and massively data-parallel computations. Taking into account that matrix manipulation programs can be more easily constructed with a visual language than with a textual language [20], Array-OL relies on a graphical formalism in which the signal processing appears as a graph of tasks. Each task reads and writes multidimensional arrays. The specification of an Array-OL application is built on two levels: a global level describes the application through a directed graph where the nodes (tasks) exchange arrays; a local level details the calculations performed on the array elements by each node. An Array-OL application directly expresses dependences between elements of arrays. In particular, temporal dependences are specified by references to elements along an infinite dimension of an array.

Lee et al. have proposed another multidimensional model to deal with such applications in [16, 5, 18, 19]. We compare their model and Array-OL in [12]. Both models have strengths and weaknesses and none includes the other. The Array-OL model is easier to understand and hides most of the complexity of scheduling. This complexity is handled through code transformations that we describe in section 4.2.

In order to allow a distributed execution of Array-OL applications, we study here how this specification model can be projected on a distributed computation model, namely the Kahn Process Network computation model [13, 14]. Distributed executions of systematic signal processing applications are useful because these applications are computation intensive. Thus using the computation power of several computers helps to reduce the execution or simulation time of such applications. Furthermore, they are often embedded and executed on parallel architectures such as Systems-on-Chip or multiprocessors. The projection we propose allows to choose the scheduling of the application and to adapt it to the target architecture.

In section 2 we explain the Array-OL model. Then in section 3, we recall the Kahn process network model of computation and describe the distributed implementation we have used. We then study how to project Array-OL specifications onto process networks and the Array-OL code transformations that allow this projection in section 4. We show an experiment on a sonar application in section 5 and finally conclude in section 6.

2 Array-OL Model of Specification

In this section we briefly sketch the Array-OL modeling language. It is important to notice that Array-OL is only a specification language, no rules are specified for executing an application described with Array-OL.

The basic principles underlying 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.

2.1 Global model

The global model 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.

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.

2.2 Local model

The local model is a little bit more complicated, it allows to express data parallel repetitions. At this level, we specify how the array elements are consumed and produced in a task. These elements are treated in parallel 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 for each repetition. That's why we call a block of data a *pattern*. In order to allow a hierarchical construction, the patterns are themselves arrays.

In order to give all the information needed to create these patterns, we need the following information:

- O: the origin of the reference pattern (for the reference repetition)
- D: the shape (size of all the dimensions) of the pattern
- P: a matrix called the paving matrix that describes how the patterns tile the array
- F: a matrix called the fitting matrix that describes the shape of the tile (how to fill a pattern with array elements)
- M: the shape (size of all the dimensions) of the array

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

2.2.1 How to fill a pattern?

From a reference element 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. Actually this matrix is a set of vectors: a vector is associated to each dimension of the pattern.

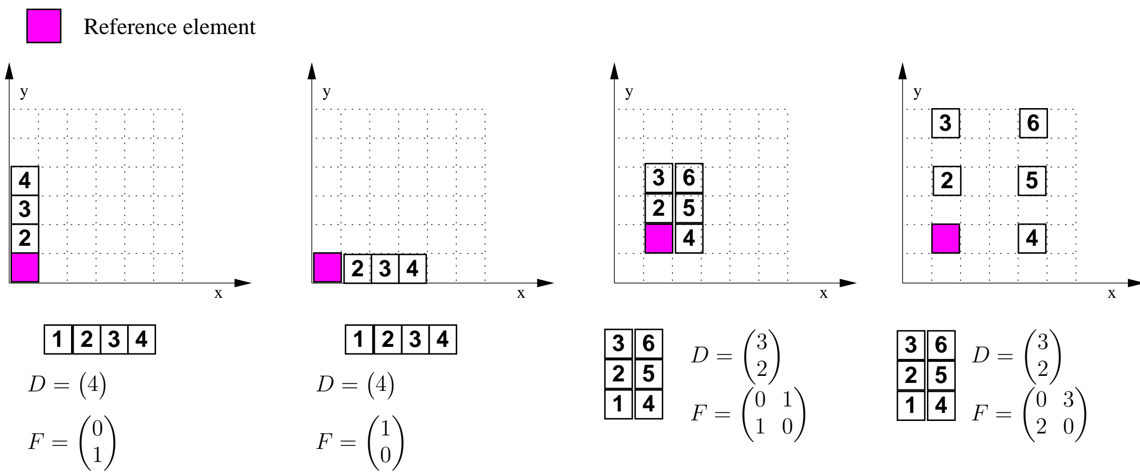

Figure 1: Construction of a pattern

The coordinates of the elements of the pattern are built as the sum of the coordinates of the reference element and a linear combination of the fitting vectors. Matrix D gives the bounds of the linear factors on each dimension of the fitting matrix. Figure 1 gives several examples of fitting matrices and patterns.

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

2.2.2 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 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.

Figure 2 shows some examples. On this figure the reference elements of the pattern have a number to distinct 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.

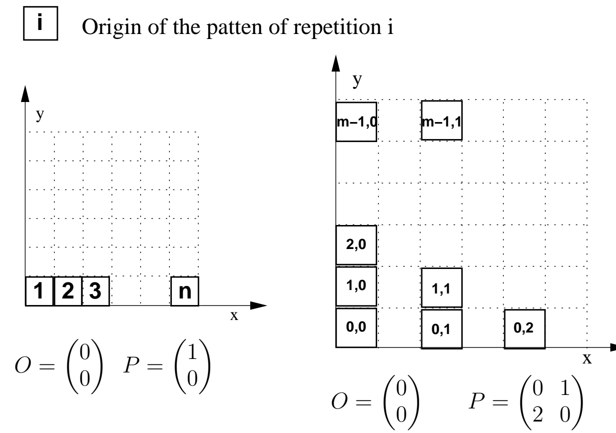

Figure 2: Origin points of a pattern

2.2.3 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 the vector Q which contains the bounds of repetition for each vector of the paving matrix.

2.2.4 Summary and non trivial examples

We can summarize all these explanations with two formulas:

- $\forall X_q, 0 \leq X_q < Q, (O + P \times X_q) \mod M$ give all the reference elements of the patterns, Q being the shape of the repetition domain.
- $\forall X_d, 0 \leq X_d < D, (O + P \times X_q + F \times X_d) \mod M$ enumerates all the elements of a pattern for the X_q repetition.

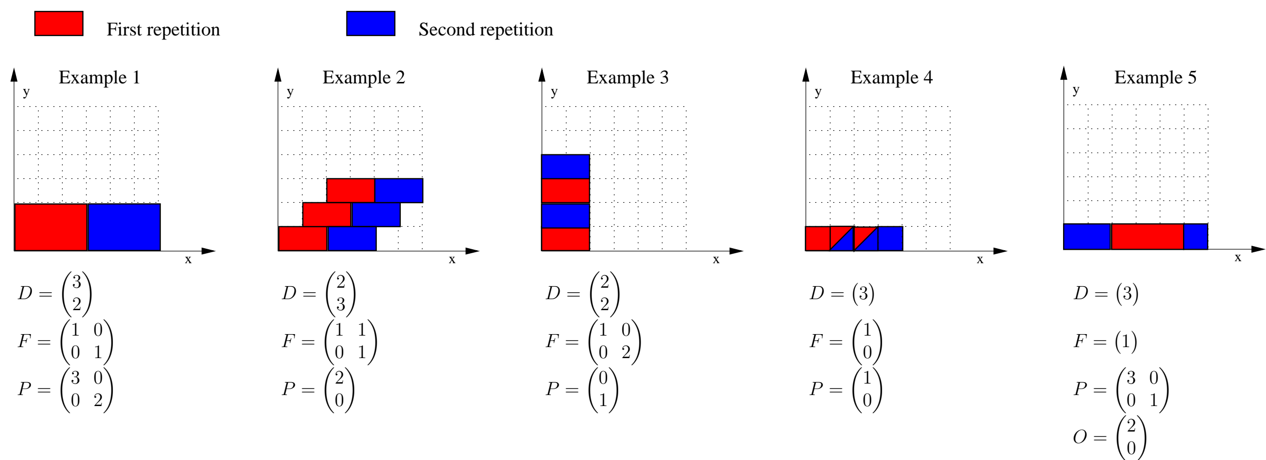

Figure 3: Different repetitions

In order to show the very interesting possibilities of the local model, we will illustrate it with a few examples: all the examples of figure 3 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 always distinct.
5. The array are toroidal and the origin here is not $\begin{pmatrix}0\\0\end{pmatrix}$ but $\begin{pmatrix}2\\0\end{pmatrix}$.

2.3 Hierarchy

According to the specification of the local model, Array-OL applications must respect some obligation on the pattern shape: they are themselves arrays. This allows to build an application in a hierarchical way. Indeed the computation made in a local model can be done not only by an atomic task like a fast Fourier transformation but by an 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.

2.4 Summary

Array-OL allows us to describe systematic signal processing applications in a very convenient way. More precisely we describe data 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 does not impose any execution order. We will see below how a distributed execution of Array-OL specifications can be obtained by projection onto the Kahn process network model of computation.

3 Kahn Process Network Model of Computation

3.1 Model

The process network model has been proposed by Kahn and MacQueen [13, 14] to easily express concurrent applications. Processes communicate only through unidirectional FIFO queues. A process is blocked when it attempts to read from an empty queue. A process can be seen as a mapping from its input streams to its output streams. The number of tokens produced and their values are completely determined by the definition of the network and do not depend on the scheduling of the processes. Thus the process network model is called determinate.

The choice of a scheduling of a process network only determines if the computation terminates and the sizes of the FIFO queues. Some networks do not allow a bounded execution. Parks [21] studies these scheduling problems in depth. He compares three classes of dynamic scheduling: data-driven, demand-driven or a combination of both with respect to two requirements:

1. Complete execution (the application should execute completely, in particular if the program is non-terminating, it should execute forever).

2. Bounded execution (only a bounded number of tokens should accumulate on any of the queues).

These two properties are shown undecidable by Buck [4] on Boolean data-flow graph which are a special case of process networks. Thus they are also undecidable for the general case of process networks. Data-driven schedules respect the first requirement, but not always the second one. Demand-driven schedules may cause artificial deadlocks. A combination of the two is proposed by Parks [21] to allow a complete, unbounded execution of process networks when possible.

3.2 Implementation

Several implementations of process networks are used for different purposes: for heterogeneous modeling with PtolemyII [17], for signal processing application modeling with YAPI [8] and for meta-computing in the domain of Geographical Information Systems with Jade/PAGIS [24, 25]. Only the Jade/PAGIS implementation is distributed. The PtolemyII and YAPI implementations use threads to represent the different processes.

The implementation we have used in our experiment is a distributed implementation on top of CORBA [3, 2]. One of the goals of this implementation is to hide the complexity of building distributed applications to the programmer, typically a non computer science specialist. The user should just have to write his domain specific processing functions and their prototypes and a code generator should take these specifications to produce distributed code, effectively hiding all the details of communication and synchronization, thus achieving a high level of transparency.

To reach this goal, the authors have made some restrictions on the processes in their implementation. These restrictions simplify the scheduling of the process network while retaining the expressing power needed for the applications. In this implementation processes are functional, meaning that they work on the following schema:

1. optional initialization phase where the process can write to its output queues (to allow cyclic process networks),
2. infinite loop:
   (a) read the inputs form the input queues,
   (b) compute the outputs,
   (c) store the results in the output queues.

When a process reads from an input FIFO queue, it can read (*get*) several tokens at a time and remove (*take*) another number of tokens. This is a common extension to the process network model that can be expressed easily by the original model.

These restrictions allow to easily represent complex applications based on an assembly of components. This coarse grain view of the application is better suited to a performant execution on a network of computers than a finer grain view which generates too much communications. This does not forbid the component to be parallel and to execute on a parallel computer. Basically, this model is well suited to model computation intensive meta-applications.

4 Projection of Array-OL onto Kahn Process Networks

The Array-OL specification model allows many execution orders. Actually, any execution order compatible with data dependences expressed by the specification is valid. The benefits of using a Kahn process network computation model as a foundation to execute Array-OL specifications are:

- The full parallelism of the specification can be exploited on distributed execution platforms.

- Determinism (main property of Kahn process networks) that gives good debugging and profiling possibilities.
- Simplified synchronization handling by using FIFOs.
- Systematic construction (see below).
- Easy handling of the hierarchy of the specification (see below).

4.1 From Arrays to Streams

The main question when projecting Array-OL is what kind of data structure is carried by the tokens: arrays or patterns? Indeed, a global model of Array-OL can be seen as a process network with the processes being the data-parallel tasks defined by the local models.

4.1.1 Streams of Patterns.

The first idea is usually to make a stream of patterns between processes. These processes thus take a set of patterns on each of their input to produce a set of patterns on their output.

The problem we encounter here is that arrays may be produced and consumed in different ways – composed of different pattern sets. Only in special cases are the arrays produced and consumed by the same patterns. One thus generally has to group some producing patterns in a token that is split into a group of consuming patterns by the following process. Determining such groupings is a difficult task that is at the heart of the difficulty to schedule Array-OL specifications. It has been studied as part of the fusion code transformation that will be presented in section 4.2.

Once the tokens have been determined, one still has to choose an execution order for the producing and consuming tasks that allows to pipeline these two tasks. Indeed, the arrays may be large, or even infinite, so pipelining is necessary to ensure a “reasonable” execution. What we mean be “reasonable” is an execution that does not loop infinitely on a given subtask, that does not use unnecessarily large amounts of memory and that does not compute too many times the same intermediate data. This is already difficult when considering two tasks but one needs to pipeline a directed graph of tasks (or process network).

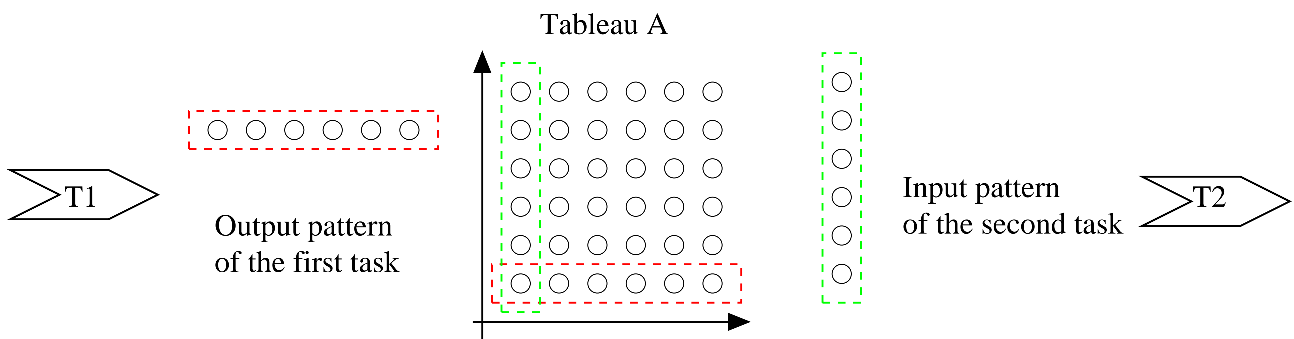

Figure 4: Corner turn

Figure 4 shows are a typical example illustrating these difficulties: the “corner turn” where an array is produced by rows and consumed by columns.

4.1.2 Streams of Arrays.

The other alternative is to make streams of arrays. One has to look at an Array-OL specification at another level: The main level of the hierarchical expression is now the local model (the data-parallel

repetition). If the repetitive task is hierarchical, this task is described by a global model itself that can be seen as a process network.

The repetition (local model at depth l of the hierarchical specification) generates a number n of data-parallel repetitions of computations of the process network (global model at depth $l+1$). The order of execution of these repetitions is not specified and can be chosen at will. Instead of having n instances of each task at level $l+1$ exchanging 1 array, one can have 1 instance of each of those tasks working on a stream of arrays. The only thing that has to be done carefully is filling the input array streams and storing the output array streams in a consistent way. This is easy to do by choosing an enumeration order of the repetition space described by the paving at level l. Thus an array at level l is transformed into a stream of arrays at level $l+1$ (or patterns of level l).

This projection of Array-OL onto process networks can be summarized as follows:

- "Array" $\mapsto$ "token".
- "Elementary task" $\mapsto$ "process".
- "Local model data-parallel repetition" $\mapsto$ "stream".

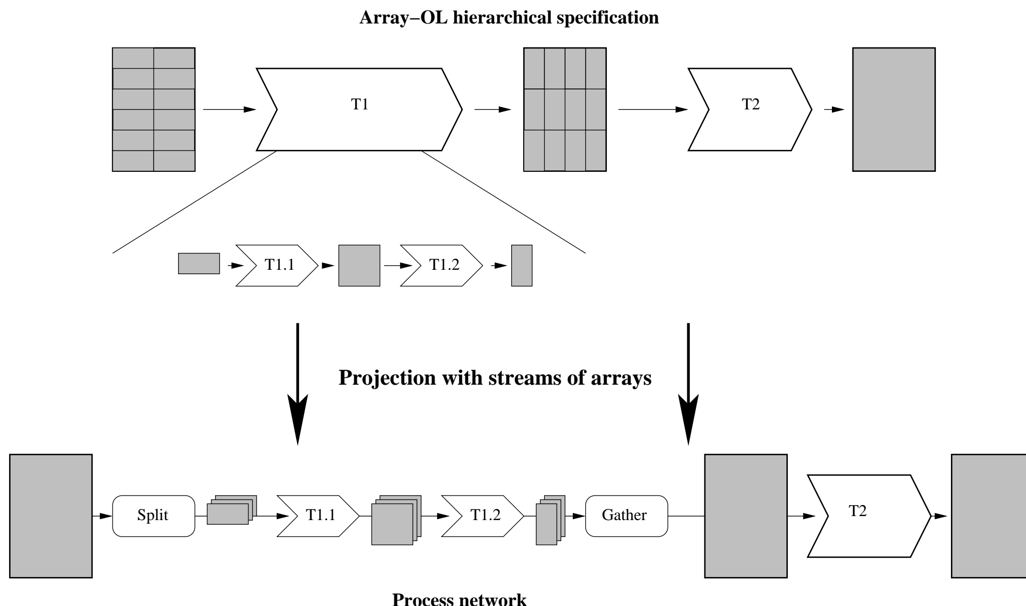

Figure 5: Example of the projection of a hierarchical Array-OL specification onto a process network using streams of arrays.

When dealing with hierarchical Array-OL specifications, one may have to add some processes in the network as illustrated by figure 5. Indeed at a given top level, a repetitive task reads full array tokens and outputs full array tokens. If the repetitive task is itself defined as a global model, this bottom level global model can also be transformed in the same way in a process network exchanging array tokens. These bottom level arrays are (at least for the input and output ones) top level patterns. So the top level arrays have to be transformed into bottom level arrays or top level patterns. This can be done by adding splitting processes reading input array streams and writing input pattern streams and gathering processes reading output pattern streams and writing output arrays. These splitting and gathering processes must scan the repetition domain in the same order. This can be done easily automatically.

Such a translation has the advantage that it is direct and systematic. The only choice is the order of the patterns of level l in the streams. The difficulty comes from the fact that the time and space

dimension are uniformized. So, to choose an order that is coherent with the availability of data at the input of the application, one has to ensure an order compatible with the flow of time.

From there comes the main drawback of this projection: it is not always possible to find such an order! Indeed, if the top level of the application is a global level manipulating infinite arrays, there is no way to produce a stream because there is no surrounding repetition. If we apply the projection, the first task reads a token that is an infinite array and no computation is ever completed.

This problem has already been observed when trying to execute Array-OL applications on multi-threaded workstations [23, 22]. The solution that has been proposed is to transform the application in order to create a new hierarchical level where the infinite arrays only appear at the top level and only one task reads from and writes into such infinite arrays. This task is itself repetitive and can thus be transformed into a stream of finite arrays. Such a stream is the usual implicit repetition of data-flow formalisms.

In the following section, we will describe the available Array-OL transformations that can be used to allow the projection we have described here.

4.2 Array-OL Transformations

As shown in section 2, the Array-OL model has some particularities that prevent the use of general loop transformations like loop fusion, distribution or unimodular loop transformations [7, 1]. These particularities are the systematic presence of the modulo operator in the paving and fitting expression and the possibly infinite size of one of the dimensions of the arrays. On the other hand, the fact that the Array-OL formalism respects the single assignment form [6] and that the repetitions are completely parallel simplifies some transformations.

In [23, 22], the authors have introduced an ad-hoc formalism (the ODT, array distribution operators – *Opérateurs de Distribution de Tableaux* in French) to transform Array-OL specifications in order to make the application executable with a naive execution scheme. In [11], the *fusion* transformation has been completely proved and generalized to more complicated cases. This fusion is the building block of nearly all the other transformations that are needed for the projection of Array-OL onto Kahn process networks. We will first explain this transformation and its interest for this study and then show the other useful transformations.

4.2.1 Fusion of two repetitive tasks

The fusion aims at reducing two tasks in a single one. This new task is hierarchical and calls the two original tasks as sub tasks.

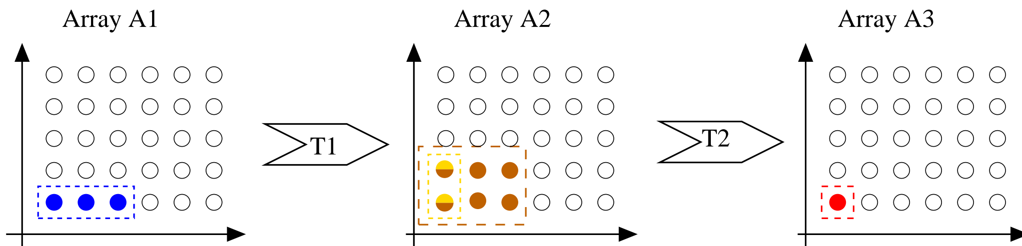

Figure 6: Before fusion

Figure 6 shows an example of two tasks to be fused. Array $A2$ is produced by task $T1$ by the way of patterns of two elements and consumed by task $T2$ by patterns of 6 elements. Figure 7 shows the result of the fusion. The new task $T3$ realizes in one step what was computed by tasks $T1$ and $T2$ .The inputs and outputs are kept identical. Thus the fusion is a local transformation that does not disturb the rest of the application. The new subtasks $T1'$ and $T2'$ compute the same operations as

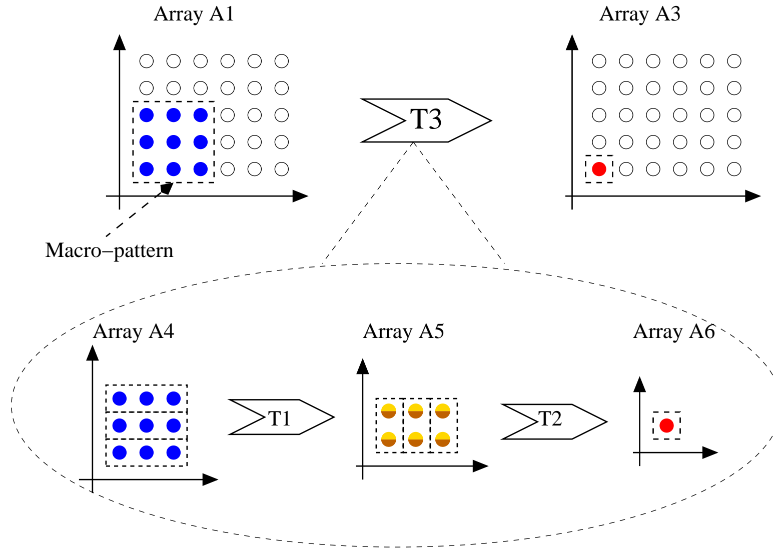

Figure 7: After fusion

$T1$ and $T2$ but on different inputs and produce different outputs. These new inputs and outputs are sub-arrays of the arrays which $T1$ and $T2$ were connecting. They are called macro-patterns and are constituted of an whole number of patterns of the original repetitions in $T1$ and $T2$.

The computation of the fusion in the general case (when the fittings and pavings are non parallel to the axes, present some shifting, tile some arrays in a cyclic way or are non compact) is challenging. The complete description of this process is beyond the scope of this paper and is available in [11]. A complete implementation of this transformation has been realized in the Gaspard project [15]. This transformation has been extended to several useful cases such as when the tasks have several inputs or outputs or even when they are connected by several intermediate arrays.

Fusing two tasks this way allows to build a hierarchy level that can then be used to make a stream of arrays and then allow the successful projection of the transformed application onto a process network.

4.2.2 Other Array-OL transformations

The fusion alone is not sufficient. Indeed, this transformation has a number of drawbacks:

- In some cases it may produce a situation where some computations are done several times.
- The chaining of several fusions produces a new hierarchical level for each, leading to very deep hierarchies.
- It is not always optimal concerning the size of the intermediate arrays. This is however borderline.
- Finally in border cases, the result may not be amenable to another fusion because it does not fully compliant to the Array-OL model. This situation is very uncommon and we have never met it in real cases.

To alleviate the first two problems, two other transformations have been proposed in [22]: the “change paving” and the “one level” transformations. The change paving increases the size of the macro-pattern in a way to reduce the repeated computations.

Figure 8 shows the result of the fusion of two tasks where the patterns of the second iteration overlap. This pattern overlap before fusion leads to elements that are shared between different repetitions of the intermediate array $A5$ after fusion. In the figure, two repetition of the top level task $T3$ are shown, one with dark colors and the other with light colors. The elements shared by the patterns of

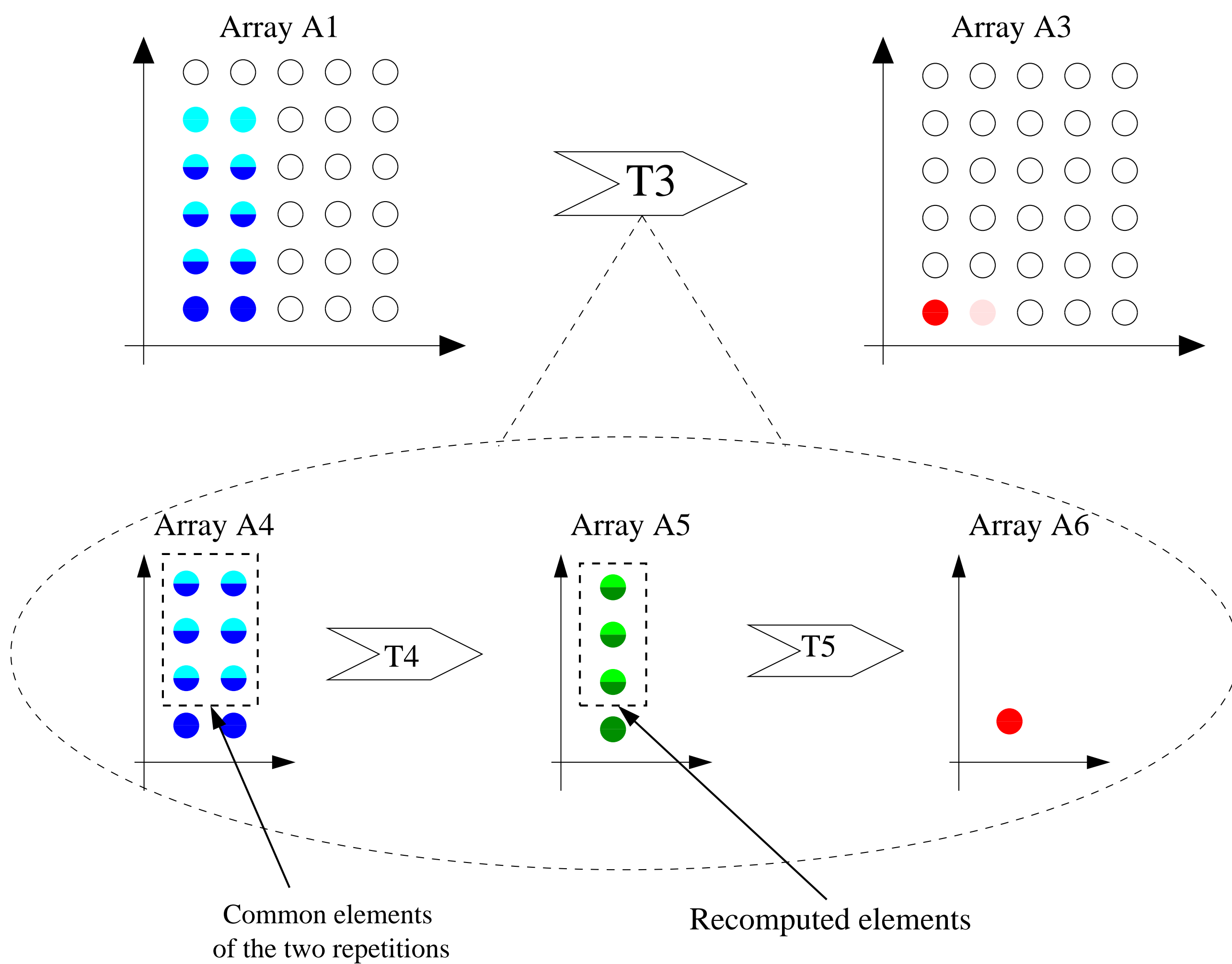

Figure 8: Before change paving

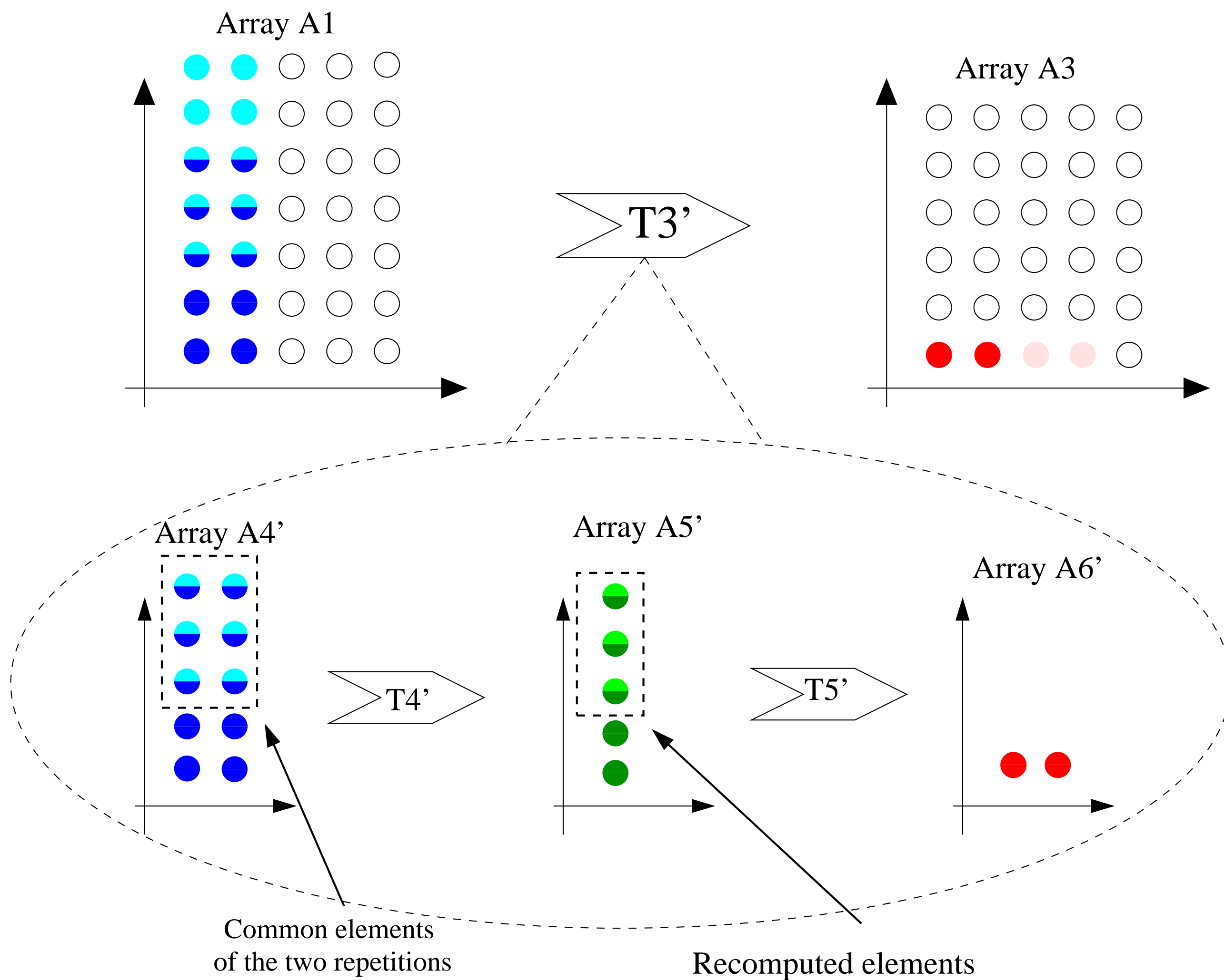

Figure 9: After change paving

the two repetitions have the two shades. Here 3 elements are shared in $A5$ and thus are computed twice. To reduce this overhead, one can increase the size of the macro-pattern. An example of this is shown in figure 9. Once again 3 elements are shared by the two repetitions. But, as there are half the number of repetitions (each one produces two points instead of one), the number of re-computations is halved.

An extreme case of the change paving transformation leads to a repetition with no more repetition. The patterns become the full arrays. In that case, one can remove the hierarchy. This becomes handy to deal with chains of fusions. A sequence of fusions and change pavings that results in the fusion of a sequence of tasks with the creation of a sole hierarchy level is called the "one level" transformation. That transformation can be used to deal with the problem of an application whose top level is a global model whose arrays have an infinite dimension. In that case, the transformed application exhibits a top level repetitive task that infinitely repeats a finite global model. That infinite repetition can then be projected as a stream and the global model as a process network.

A final useful transformation is the "nesting" where the repetition domain is split in two nested repetitions. It consists in the creation of a hierarchical level whose global model is a single repetition whose patterns (called macro-patterns) are unions of the original patterns. These macro-patterns are then consumed by a second nested repetition that works with the original patterns. This transformation allows to adapt the granularity of the application.

4.3 Projecting Array-OL Specifications onto Distributed Process Networks

The above transformations can be used together to transform the specification into another one that expresses the same dependences between the array elements but that is more suited to be executed on a given platform. The kind of platforms we focus on in this article is an heterogeneous distributed architecture (as a network of workstations or a System-on-Chip).

There are two ways one could benefit from the regular parallelism exhibited by a data-parallel repetitions:

1. A SPMD execution on several execution units (or using several threads).

2. The transformation of that repetition in a stream as expressed in section 4.1.2.

Using a third possibility, namely a sequential execution, one can propose a large family of schedules for a given specification by tagging each data-parallel repetition by the execution strategy: SPMD, Process Network or Sequential.

Selecting the "best" solution, both in terms of specification transformation and schedule choice is way beyond the focus of this paper and is a research interest we will pursue in future research.

5 Experiment

We present here an experiment consisting in the distributed implementation of an Array-OL specification with the distributed process network runtime presented in section 3.2.

The initial Array-OL specification is the beginning of a sonar application. It consists in two repetitive tasks handling four arrays: a $(512 \times \infty)$ array as the input of the first task, a $(512 \times 256 \times \infty)$ array as the output of the first task and input of the second one, a $(128 \times 200 \times 192)$ coefficient arrays as the second input of the second task and finally a $(128 \times \infty \times 200)$ array as the output of the second task.

After fusion of the two tasks, the infinite dimension is split in 512-element chunks and the application can be projected onto a process network with two processes. The first process takes as input tokens (512×512) arrays, the two processes exchange (512×256) arrays and the output of the second process is a stream of (128×200) array tokens. The coefficient array can be included in the second process as it is read completely by each repetition of the second task.

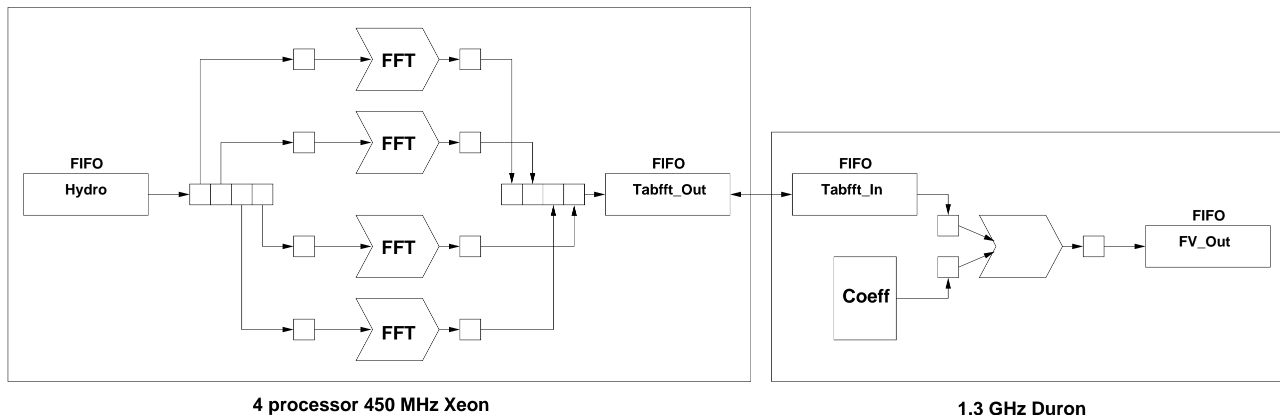

Figure 10: Example distributed process network

This application will be deployed on two workstations on the same Ethernet network. As one of them is a 4-processor SMP computer, we have used the nesting transformation on the first task to exhibit four threads. The resulting process network is described on figure 10. The first task is mapped as 4 parallel processes (implemented as treads) on a 4-processor 450 MHz Intel Xeon and the second is a 1.3 GHz AMD Duron.

Table 1: Performance measures

Stream length	Total time (s)	Task1 (s)	Task2 (s)	Effic.1	Effic.2
8	17	13	12	0.76	0.70
16	30	26	24	0.86	0.80
24	44	40	36	0.90	0.81
32	58	53	49	0.91	0.84
40	71	66	62	0.92	0.87
48	85	80	75	0.94	0.88
56	99	94	88	0.94	0.88
64	113	107	101	0.94	0.89
72	126	120	115	0.95	0.91
80	139	133	128	0.95	0.92

Table 1 shows the obtained performances. This tables shows the execution times of each task (measuring only the time to compute the elementary tasks) and of the complete distributed application in seconds measured with different token stream lengths. As expected, the computation time grows linearly with the length of the token stream and the total execution time is a little bit more than the time of the slowest of the two tasks. The efficiency measured as the proportion of the computation time of each computer during the execution of the complete application is very high. That means that the proposed projection of the application onto the architecture is well adapted. The set of available transformations of the original Array-OL specification has allowed us to propose such a good mapping on this particular application. A different transformation would be used on a different architecture.

6 Conclusion

We have presented in this paper the Array-OL specification model. This multidimensional model allows to fully express the potential parallelism of systematic signal processing applications. This specification is completely indepentent on the execution architecture. We have shown how several

code transformations have been implemented to allow to derive a form of the application that can be projected directly onto a process network.

This projection allows a distributed execution of Array-OL applications onto heterogeneous distributed architectures such as Systems-on-Chip or networks of workstations.

In future work, we will study heuristics to select the chain of transformations that leads to the "best" schedule of the application on a target architecture, taking into account real time constraints, power consumption, cost, etc.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Oct. 2001. `http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-286-0`.

[2] A. Amar. *Support d'exécution pour le metacomputing à l'aide de CORBA*. PhD thesis, Université des sciences et technologies de Lille, Laboratoire d'informatique fondamentale de Lille, Dec. 2003. (In French).

[3] A. Amar, P. Boulet, J.-L. Dekeyser, and F. Theeuwen. Distributed process networks using half FIFO queues in CORBA. In *ParCo'2003, Parallel Computing*, Dresden, Germany, Sept. 2003.

[4] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California at Berkeley, 1993.

[5] 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.

[6] J.-F. Collard. *Reasoning about program transformations: imperative programming and flow of data*. Springer-Verlag, 2003. ISBN 0-387-95391-4.

[7] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000. `http://www.birkhauser.com/detail.tpl?isbn=0817641491`.

[8] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, CA, June 2000. ACM Press.

[9] 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.

[10] 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.

[11] 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.

[12] P. Dumont and P. Boulet. Another multidimensional synchronous dataflow, simulating Array-OL in PtolemyII. to appear, 2005.

[13] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland, Aug. 1974.

[14] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77: Proceedings of the IFIP Congress 77*, pages 993–998. North-Holland, 1977.

[15] Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille. Gaspard home page. `http://www.lifl.fr/west/gaspard/`, 2005.

[16] 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.

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

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

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

[20] R. Pandey and M. Burnett. Is it easier to write matrix manipulation programs visually or textually? An empirical study. In *IEEE Symposium on Visual Languages*, pages 344–351, Bergen, Norway, Aug. 1993.

[21] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD Thesis, EECS Department, University of California, Berkeley, CA, Dec. 1995.

[22] 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).

[23] 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.

[24] D. Webb, A. Wendelborn, and K. Maciunas. Process networks as a high-level notation for metacomputing. In *Workshop on Java for Parallel and Distributed Computing (IPPS)*, Puerto Rico, Apr. 1999.

[25] D. Webb, A. Wendelborn, and J. Vayssière. A study of computational reconfiguration in a process network. In *IDEA7*, Victor Harbour, South Australia, Feb. 2000.

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