Multidim Syst Sign Process (2010) 21:105–131
DOI 10.1007/s11045-009-0085-4

Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing

Calin Glitia · Philippe Dumont · Pierre Boulet

Received: 22 January 2007 / Revised: 11 November 2008 / Accepted: 16 March 2009 /
Published online: 31 March 2009

Abstract Intensive signal processing applications appear in many application domains such as video processing or detection systems. These applications handle multidimensional data structures (mainly arrays) to deal with the various dimensions of the data (space, time, frequency). A specification language allowing the direct manipulation of these different dimensions with a high level of abstraction is a key to handling the complexity of these applications and to benefit from their massive potential parallelism. The Array-OL specification language is designed to do just that. We introduce here an extension of Array-OL to deal with states or delays by the way of uniform inter-repetition dependences. We show that this specification language is able to express the main patterns of computation of the intensive signal processing domain.

Keywords Multidimensional signal processing · Dataflow · Synchronous dataflow · Array-OL · Data parallelism

1 Introduction

Computation intensive multidimensional applications are predominant in several application domains such as image and video processing or detection systems (radar, sonar). By multidimensional, we mean that they manipulate primarily multidimensional data structures such

C. Glitia
INRIA Lille - Nord Europe, Parc Scientifique de la Haute Borne, 40, Avenue Halley, Bât.A,
Park Plaza, 59650 Villeneuve d'Ascq, France
e-mail: Calin.Glitia@inria.fr

P. Dumont
NXP Eindhoven, Eindhoven, The Netherlands
e-mail: Philippe.Dumont@nxp.com

P. Boulet (✉)
LIFL, Université de Lille 1, Cité Scientifique, 59 655 Villeneuve d'Ascq Cedex, France
e-mail: Pierre.Boulet@lifl.fr

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 and others such as frequency dimensions can appear lately during the computation. Actually, such an application manipulates a stream of 3D arrays.

Dealing with such applications presents a number of difficulties:

- They are often massively parallel and to be able to benefit from this massive parallelism, it must be fully expressed.
- Only a few models of computation are multidimensional (and thus allow to express the full potential parallelism of the applications) and none of them are widely used.
- The patterns of access to the data arrays are diverse and complex. They are actually the main problem when trying to optimize these applications.
- Scheduling these applications with bounded resources and time is challenging, especially in a parallel and distributed context.

As we will see in Sect. 2, there has been a few attempts to design specification languages or models of computation for this application domain. The challenge for these languages is to provide a way to specify multidimensional data accesses without compromising the usability of the language and if possible provide a way to statically schedule these applications on parallel hardware platforms. The features that a good language for multidimensional intensive signal processing ought to possess are a way to access the multidimensional data structures via sub-arrays, the support of sliding windows, the possibility to deal with cyclic data accesses, the possibility to deal with several sampling rates in the same specification and some way to express delays or simple stateful computations such as recursive filters. These features should be directly available in the language to allow the programmer to express the full potential parallelism of his application.

We propose here an extension of the Array-OL model of specification to deal with state or delay modeling. We call this new language *Array-OL with delays*. After a review of the current language proposals for multidimensional signal processing in Sect. 2, we recall the bases of Array-OL in Sect. 3 and present our extension in Sect. 4.

2 Related work

Only a few models of computation (MoC) have attempted to propose formalisms to model and statically schedule multidimensional signal processing applications.

Table 1 presents a comparison of several languages (or models of computation) dedicated to signal processing. This comparison highlights the suitability of the various languages for intensive multidimensional signal processing. The main comparison criteria are the allowed data structures (mono dimensional data flows or multidimensional arrays) and the expressivity of the access functions to these data structures. The class of applications these languages are able to deal with is also constrained by the control flow mechanisms they allow. As a common point, all these languages permit static scheduling in order to build efficient implementations. We have deliberately not included the dynamic variants of SDF or general purpose parallel programming languages though some of their features could be interesting in our context.

Most of the compared languages are based on SDF (Synchronous Data Flow) or on its multidimensional extension, MDSDF (Multi-dimensional Synchronous Data Flow). A detailed comparison of MDSDF, GMDSDF and Array-OL (without delays) is available in Dumont and Boulet (2005). In Sect. 2.1 we make a more pragmatic comparison between

Table 1 Suitability of various models of computation for intensive signal processing

MoC	Data type	Access type	Access generality			Control structures Delays
			Sliding windows	Sub/over sampling	Non // to the axes	
SDF (Lee and Messerschmitt 1987a,b)	1D	sub-array	–	–		+
CSDF (Bilsen et al. 1995)	1D	sub-array	–	–		+
MDSDF (Lee 1993; Chen and Lee 1995)	mD	sub-array	–	+	–	+
GMDSDF (Murthy 1996; Murthy and Lee 2002)	mD	sub-array	–	+	+	+
WSDF (Joachim Keinert and Haubelt 2005; Keinert et al. 2006)	mD	sub-array	+	+	–	+
Array-OL (Demeure et al. 1995; Demeure and Del Gallo 1998)	cyclic mD	sub-array	+	+	+	–
Array-OL with delays	**cyclic mD**	**sub-array**	**+**	**+**	**+**	**+**
Stream-IT (Thies et al. 2002)	1D	sub-array	+	+		+
Alpha (Le Verge et al. 1991; de Dinechin et al. 1995)	polyhedra	affine	+	+	+	+
Sisal (Gaudiot et al. 2001; Attali et al. 1995)	mD	sub-array	+	+	–	+
SaC (Scholz 2003)	mD	sub-array	+	+	+	+

Concerning the data type column, 1D means that the scheduling considers mono-dimensional data streams (that may carry multidimensional arrays as in StreamIT), mD means that these data streams are replaced by multidimensional arrays, cyclic mD means that some dimensions of these multidimensional arrays may be cyclic and polyhedra means that the language handles convex polyhedra of integer points. A "+" in a column means that the feature is supported, a "–" that it is not

the Array-OL model and the class of SDF models, including the recently proposed WSDF (Windowed Synchronous Data Flow) model. We then present in Sect. 2.2 some other langages that have been proposed for various purposes and that have some nice features for the intensive signal processing domain.

The goals of Array-OL and the class of models based on multi-dimensional SDF are similar and although 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.

As can be seen in the table, Array-OL with delays can easily deal with all the following requirements of the application domain:

- *Access to multidimensional arrays by regularly spaced sub-arrays*: Intensive multidimensional signal processing is characterized by extremely regular and parallel data treatment. In order to fully express all the characteristics of such applications, it is essential to completely and correctly describe all this regularity.

- *Ability to deal with sliding windows*: Sliding window algorithms are fundamental parts of signal processing systems. Many of the well known data flow models have difficulties manipulating key concepts for describing such applications, like multiple data consumption and border processing.
- *Ability to deal with cyclic array dimensions*: In the case of some applications where certain spatial dimensions may represent physical tori—like hydrophones around a submarine—the ability to specify cyclic array dimensions is crucial. Also after a discrete Fourier transform, the frequency dimension is cyclic. To deal with bands of frequencies—as when removing some jamming by nullifying a band of frequencies in a software radio application—the ability to address in a cyclic way a range of indices is very important.
- *Ability to sub/over sample the arrays*: Application constrains may cause certain components to consume arrays that represent sub/over samples of the available arrays. Thus the MoC should be able to deal with different rates of production or consumption of the data.
- *Hierarchical specification to deal with complex systems*: A hierarchical specification is mandatory to model complex applications. It favors modularity and component reuse while allowing the structuring of the application at different granularity levels.
- *Expression of delays*: Delays and self-loops are needed to allow tasks to maintain state information in a data-flow language (for example to deal with recursive filters).

SDF allowing hierarchical contructions implies that all the extensions also allow such constructions.

The possibility to define sub-arrays that are not parallel to the axes is possible with Array-OL and GMDSDF though it may not be necessary for the multidimensional signal processing domain. In both cases it is a consequence of the generality of the approach. The control flow mechanisms are very limited in Array-OL. More complex mechanisms based on mode automata have been proposed by Labbani et al. (2006, 2005) and are under implementation in Gaspard2 (DaRT Team LIFL/INRIA 2008).

2.1 SDF and its extensions

2.1.1 SDF

The SDF (Synchronous Data Flow) model was created and developed by Edward A. Lee since 1986 in order to model simple dataflow systems. Lee has integrated the model in his well-known modeling and simulation environment for embedded systems: Ptolemy. In SDF, an application is described as an oriented graph; each node consumes data on its incoming edges and produces data on its outgoing edges, edges representing one-dimensional data streams. In order to make an SDF application statically defined, two restrictions were introduced to the SDF model:

- first, any node (named "actor" in Ptolemy) always consumes and produces the same amount of "tokens" (data elements) at each execution, amount known at modeling time;
- secondly, the values of data cannot influence the control flow.

The static definition property associated to SDF applications implies extremely interesting consequences. It makes it possible to schedule the application at modeling time, to have a deterministic execution, to detect deadlocks and to execute the application with the use of a bounded amount of memory.

The delays are another important characteristic of the SDF model. 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 causes the consumption task to initially take the values already present on the edge before consuming the result of the production task.

Delays allow SDF to express cyclic graphs and an eventual blockage in the specification can be statically detected.

A state is a special delay, expressed with an self-loop with delay on an actor, which allows the actor to use the result of a certain iteration in a future one. Of course for the first iterations there must be a default value, same as with delays. With a state, it is possible to describe mathematical operations like the discrete integration or IIR filters.

Applications described with SDF have the major advantage of being statically defined, but the restrictions on the model together with the mono-dimensionality aspect drastically reduce the set of applications that it can be describe. There are two directions of evolution for the SDF model in order to extend its domain of applicability. The first is the introduction of multi-dimensionality explored with the Multi-Dimensional SDF(MDSDF) and its extensions (GMDSDF, WSDF) and the second direction is by playing with the restrictions, by allowing parameterized token manipulation (parameterized dataflow) or the introduction of functionality modes when modeling an application (Cyclo Static Data Flow, Scalable Synchronous Data Flow, Blocked Data Flow). What we must note is that all these evolutions, being designed to be compatible with the SDF model and in the same time keep as much as possible the static definition property, tend to get extremely complicated to manipulate (as we will see later the case of GMDSDF).

In this paper we are interested only in the first type of extension, the one based on the introduction of multi-dimensionality; the second is similar in goal with the control extension of Array-OL mentioned earlier. 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. This type of multidimensionality is not appropriate for representing and manipulating complex multidimensional applications where some data arrays may be produced and consumed in different way, the classical example is the corner-turn where a 2D array is produced in rows and consumed in columns.

2.1.2 MDSDF

MDSDF was first introduced to provide a way 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. Although some MDSDF applications can still be modeled in SDF by a linearization of the dimensions, the description is more complicated to understand and for most of real multi-dimensional applications it is impossible to specify in SDF how to build the precedence graph.

2.1.3 GMDSDF

The MDSDF model allows the use of multidimensional streams but the consumption and production of data is restricted to be parallel with the axes. In an attempt to remove these restrictions, Praveen K. Murthy and Edward A. Lee have proposed another extension named GMDSDF (Generalized Multidimensional Synchronous Dataflow). The idea is to produce points on a lattice having a form decided by a special actor named source and that can be modified by two other actors (decimator and expander). More details on how these actors

can be found in Murthy and Lee (1995). What is important to point-out is that the extensions introduced in GMDSDF made it extremely difficult to solve the system of equations needed to compute the static scheduling. In order to compute this scheduling a lot of simplifications are needed, described by Lee only for 2-D applications. Although GMDSDF is an extension of MDSDF, it seems that the most convenient way to compute the scheduling is to consider the GMDSDF applications as those of MDSDF (the simplifications proposed represent reductions to generalized rectangles).

As Murthy and Lee said in their paper (Murthy and Lee 2002): "the definition of a GMDSDF will not be easy to use in a programming environment" and they have not implemented the model in PtolemyII. In our opinion, one of the problems is that only the expander and the decimator can change the so-called 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 has to be very regular and does not allow to take care of more complicated patterns of data access.

2.1.4 Windowed synchronous data flow

Windowed synchronous data flow is a recent model of computation based on MDSDF elaborated specially to deal with algorithms having *sliding window* functionalities, an important part of any image processing system. As major lacks in previous data-flow models we can mention the impossibility of having multiple data consumption and border processing. This model abstracts important characteristics of sliding window algorithms and leads to more precise representations but all this by the use of new concepts like virtual tokens, effective tokens, virtual token union, window translation, border processing, etc. All the additional concepts make the modeling extremely complicated and might even lead to ambiguities. The model has the advantage of keeping the static definition property and it can be used as a basis to solve many important questions about scheduling and buffer estimation of many important static image processing algorithms but this with the cost of overcomplicating the specifications and restricting the applicability of the model to sliding window algorithms.

2.2 Other languages

2.2.1 StreamIt

We just mention the StreamIt programming language and compilation infrastructure, specifically engineered for modern streaming systems. It is designed to facilitate the programming of large streaming applications, as well as their efficient mapping.

The StreamIt vision focuses on programmability, domain specific optimizations and architecture specific optimizations. It is an attractive programming model because of a simple mapping from specification to implementation, but it has as a major draw-back its lack of expressiveness due to the manipulation of only mono-dimensional streams.

2.2.2 Alpha

An other language worth mentioning is Alpha, a functional language based on systems of recurrent equations (Karp et al. 1967). Alpha is based on the polyhedral model, which is extensively used for automatic parallelization and the generation of systolic arrays. Alpha shares some principles with Array-OL:

- Data structures are multidimensional: union of convex polyhedra for Alpha and arrays for Array-OL.
- Both languages are functional and single assignment.

With respect to the application domain, arrays are sufficient and more easily handled by the user than polyhedra. Some data access patterns such as cyclic accesses are more easily expressible in Array-OL than in Alpha.

2.2.3 Sisal

Sisal (Stream and Iteration in a Single Assignment Language) is a single assignment functional language designed with the goal of providing a general-purpose user interface for a wide range of parallel processing platforms. Its semantics define the dynamic of a data-flow graph, Sisal expressions evaluate and return values based solely on the values bound to their formal arguments and constituent identifiers. The initial Sisal definition [version 1.2 Gaudiot et al. (2001)] is based on the one-dimensional array model. In this definition arrays could be constructed by the concatenation of the inner dimension arrays to build a single one-dimension array, favoring optimizations such as vectorization. The disadvantages of this design are that arrays must be stored contiguous in memory and that the inner vectors must always have the same size.

The initial definition has been extended in Attali et al. (1995) to allow multi-dimensional semantics. To express potentially infinite sequences with non-strict semantics and to provide pipelined parallelism, streams can be designed in Sisal. In the semantics for the multi-dimensional arrays, the subarrays can have different sizes. Differently from the initial version, the second one attempts to express array accesses in a structural way rather than element by element. Placement specification can be used to map the values of a subarray into a geometrically defined subset of the original array, to describe diagonal components using a *dot* notation or an arbitrary placement of values when an array is used as vector subscript. Nonetheless, predefined subarray accesses remain parallel with the axes or diagonal and in order to express structures like cyclic accesses, subarray placement must be defined element by element.

Array-OL can express more complex subarray structures than the structural array expressions of Sisal.

2.2.4 SaC

The expressive power of Sisal does not stands out very much against that of imperative languages. SaC (Scholz 2003) (Single Assignment C) is a language that tries to integrate n-dimensional arrays with constant access time into imperative languages. It picks up on the design principles of Sisal but introduces the n-dimensional arrays as the major data structures.

An array in SaC is represented by a one-dimensional data vector which contains its elements and a shape vector which defines its structure, by specifying the number of axes (or the dimensionality) and the number of elements (index range) along each axis of the array. Using the `reshape` built-in primitive a n-dimensional array can be defined or redefined by changing its shape.

Similar to other array languages, SaC suggests using *compound array operations* which apply uniformly to all the elements or to the elements of the coherent subarrays. Programming array computations can be made decidedly more concise, comprehensible, and less susceptible to errors using *compound array operations* rather than nesting of loops that traverse array

elements with specific starts, stops and strides. These operations usually involve decomposing arrays into subarrays and re-assembling them in a different form. In SaC the chosen approach is to provide sufficiently versatile language constructs which allow to specify in a concise and efficiently executable form shape-invariant compound operations, based on the so-called WITH-loop. These constructs have the important benefit of being general-enough and all compiler optimizations concerning efficient code generation for array operations can be concentrated on this single construct.

These constructions can be used to express the subarray accesses available with Array-OL but these construction must be defined by the user if not available in the libraries.

The last three imperative functional languages presented before in addition to Array-OL and the other visual languages can express arbitrary array placements by the use of element by element accesses. Array-OL does not manipulate the indices directly but accesses the arrays through sub-arrays. On the one hand that restricts the application domain but on the other hand that makes it more abstract and more focused on the main difficulty of intensive signal processing applications: data access patterns.

3 Array-OL

As a preliminary remark, 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 as is shown in Boulet (2008) where the author describes in details the semantics of Array-OL and gives the definition of a statically schedulable kernel of Array-OL.

3.1 Principles

The initial goal of Array-OL is to give a mixed graphical-textual language to express multidimensional intensive signal processing applications. As said before, these applications work on multidimensional arrays. The complexity of these applications does not come from the elementary functions they combine, but from their combination by the way they access the intermediate arrays. Indeed, most of the elementary functions are sums, dot products or Fourier transforms, which are well known and often available as library functions. The difficulty and the variety of these intensive signal processing applications come from the way these elementary functions access their input and output data as parts of multidimensional arrays. The complex access patterns lead to difficulties to schedule these applications efficiently on parallel and distributed execution platforms. As these applications handle huge amounts of data under tight real-time constraints, the efficient use of the potential parallelism of the application on parallel hardware is mandatory.

From these requirements, we can state the basic principles that underly the language:

- All the potential parallelism in the application has to be available in the specification, both *task parallelism* and *data parallelism*.
- Array-OL is a *data dependence expression* language. Only the true data dependences are expressed in order to express the full parallelism of the application, defining the minimal partial order of the tasks. Thus any schedule respecting these dependences will lead to the same result. The language is deterministic.
- It is a *single assignment* formalism. No data element is ever written twice. It can be read several times, though. Array-OL can be considered as a first order functional language.

- Data accesses are done through sub-arrays, called *patterns*.
- The language is *hierarchical* to allow descriptions at different granularity levels and to handle the complexity of the applications. The data dependences expressed at a level (between arrays) are approximations of the precise dependences of the sub-levels (between patterns).
- The spatial and temporal dimensions are treated equally in the arrays. In particular, time is expanded as a dimension (or several) of the arrays. This is a consequence of single assignment.
- The arrays are seen as tori. Indeed, some spatial dimensions may represent some physical tori (think about some hydrophones around a submarine) and the frequency domains obtained by discrete Fourier transforms are toroidal.

The semantics of Array-OL is that of a first order functional language manipulating multidimensional arrays. It is not a data flow language but can be projected on such a language. Array-OL does not handle streams but arrays of values. In the absence of delays all the elements of these arrays could be consumed or produced in parallel. The environment or the execution platform can impose an order and a granularity on these elements thus leading to a stream but the choice of the granularity of the computation is left to the compiler (or the system engineer) and not the programmer (ex: with the same Array-OL specification, a video represented as a 3D array of pixels can be computed as a flow of frames or a flow of rows or even a flow of pixels).

As a simplifying hypothesis, the application domain of Array-OL is restricted. No complex control is expressible and the control is independent of the value of the data. This is realistic in the given application domain, which is mainly data flow. Some efforts to couple control flows and data flows expressed in Array-OL have been done in Labbani et al. (2005) but are outside the scope of this paper.

The usual model for dependence based algorithm description is the dependence graph where nodes represent tasks and edges dependences. Various flavors of these graphs have been defined. The expanded dependence graphs represent the task parallelism available in the application. In order to represent complex applications, a common extension of these graph is the hierarchy. A node can itself be a graph. Array-OL builds upon such hierarchical dependence graphs and adds repetition nodes to represent the data-parallelism of the application.

Formally, an Array-OL application is a set of *tasks* connected through *ports*. The tasks are equivalent to mathematical functions reading data on their input ports and writing data on their output ports. The tasks are of three kinds: *elementary*, *compound* and *repetition*. An elementary task is atomic (a black box), it can come from a library for example. A compound is a dependence graph whose nodes are tasks connected via their ports. A repetition is a task expressing how a single sub-task is repeated.

All the data exchanged between the tasks are arrays. These arrays are multidimensional and are characterized by their *shape*, the number of elements on each of their dimensions.[1] A shape will be noted as a column vector or a comma-separated tuple of values indifferently. Each port is thus characterized by the shape and the type of the elements of the array it reads from or writes to. As said above, the Array-OL model is single assignment. One manipulates *values* and not *variables*. Time is thus represented as one (or several) dimension of the data arrays. For example, an array representing a video is three-dimensional of shape (width of

[1] A single point, seen as a 0-dimensional array is of shape (), seen as a 1-dimensional array is of shape (1), seen as a 2-dimensional array is of shape $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, etc.

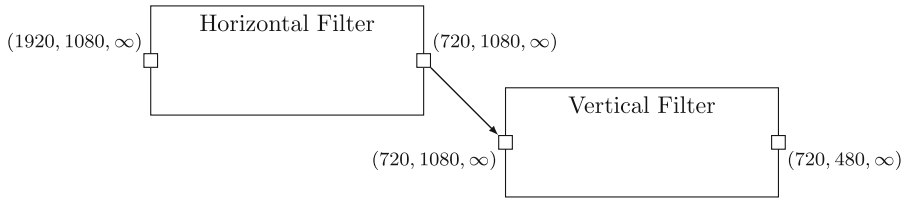

Fig. 1 Downscaler example

frame, height of frame, frame number). We will illustrate the rest of the presentation of Array-OL by an application that scales an high definition TV signal down to a standard definition TV signal. Both signals will be represented as a three dimensional array.

3.2 Task parallelism

The task parallelism is represented by a compound task. The compound description is a simple *directed acyclic graph*. Each node represents a task and each edge a dependence connecting two conform ports (same type and shape). There is no relation between the shapes of the inputs and the outputs of a task. So a task can read two two-dimensional arrays and write a three-dimensional one. The creation of dimensions by a task is very useful, a very simple example is the FFT which creates a frequency dimension. We will study as a running example a downscaler from high definition TV to standard definition TV (Fig. 1). Here is the top level compound description. The tasks are represented by named rectangles, their ports are squares on the border of the tasks. The shape of the ports is written as a t-uple of positive numbers or ∞. The dependences are represented by arrows between ports.

There is only one limitation on the dimensions: there must be at most 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.

Each execution of a task reads one full array on its inputs and writes the full output arrays. It's not possible to read more than one array per port to write 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 compound description. 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.

3.3 Data parallelism

A data-parallel repetition of a task is specified in a repetition task. The basic hypothesis is that all the repetitions of this repeated task are independent. They can be scheduled in any order, even in parallel.[2] The second one is that each instance of the repeated task operates with sub-arrays of the inputs and outputs of the repetition. For a given input or output, all the sub-array instances have the same shape, are composed of regularly spaced elements and are regularly placed in the array. This hypothesis allows a compact representation of the repetition and is coherent with the application domain of Array-OL which describes very regular algorithms.

[2] This is why we talk of *repetitions* and not *iterations* which convey a sequential semantics.

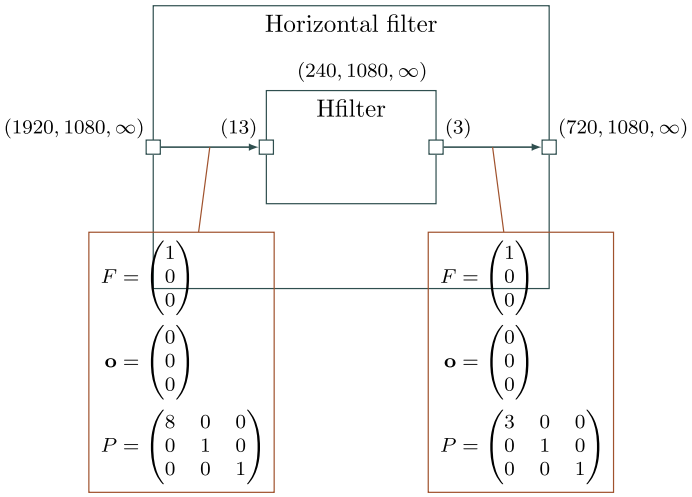

Fig. 2 Horizontal filter repetition task

As these sub-arrays are conform, they are called *patterns* when considered as the input arrays of the repeated task and *tiles* when considered as a set of elements of the arrays of the repetition task. In order to give all the information needed to create these patterns, a *tiler* is associated to each array (ie each edge). A tiler is able to build the patterns from an input array, or to store the patterns in an output array. It describes the coordinates of the elements of the tiles from the coordinates of the elements of the patterns. It contains the following information:

- F: a *fitting* matrix.
- $\mathbf{o}$: the *origin* of the *reference pattern* (for the *reference repetition*).
- P: a *paving* matrix.

3.3.1 Visual representation of a repetition task

The shapes of the arrays and patterns are, as in the compound description, noted on the ports. The *repetition space* indicating the number of repetitions is defined itself as an multidimensional array with a shape. Each dimension of this repetition space can be seen as a parallel loop and the shape of the repetition space gives the bounds of the loop indices of the nested parallel loops. An example of the visual description of a repetition is given in Fig. 2 by the horizontal filter repetition from the downscaler. The tilers are connected to the dependences linking the arrays to the patterns. Their meaning is explained below.

3.3.2 Building a tile from a pattern

From a *reference element* (**ref**) in the array, one can extract a pattern by enumerating its other elements relatively to this reference element. The *fitting* matrix is used to compute the other elements. The coordinates of the elements of the pattern ($\mathbf{e_i}$) are built as the sum of the coordinates of the reference element and a linear combination of the fitting vectors as follows

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{ref} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}} \quad (1)$$

where $\mathbf{s}_{\text{pattern}}$ is the shape of the pattern, $\mathbf{s}_{\text{array}}$ is the shape of the array and F the fitting matrix.

In the following examples of fitting matrices and tiles, the tiles are drawn from a reference element in a 2D array. The array elements are labeled by their index in the pattern, $\mathbf{i}$, illustrating the formula $\forall\, \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{ref} + F \cdot \mathbf{i}$. The fitting vectors constituting the basis of the tile are drawn from the reference point.

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 the coordinates of the tile elements are computed modulo the size of the array dimensions. The following more complex examples of tiles are drawn from a fixed reference element ($\mathbf{o}$ as origin in the figure) in fixed size arrays, illustrating the formula $\forall\, \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{o} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}}$.

3.3.3 Paving an array with tiles

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

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

$$\forall\, \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \mathbf{ref_r} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{array}} \tag{2}$$

where $\mathbf{s}_{\text{repetition}}$ is the shape of the repetition space, P the paving matrix and $\mathbf{s}_{\text{array}}$ the shape of the array. Here are some examples (Figs. 3–8).

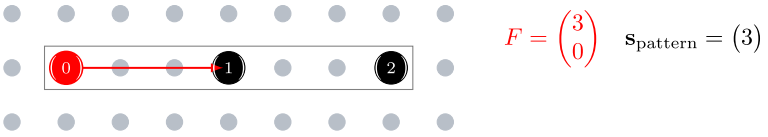

Fig. 3 Sparse tile. There are here 3 elements in this tile because the shape of the pattern is (3). The indices of these elements are thus (0), (1) and (2). Their position in the tile relatively to the reference point are thus $F \cdot (0) = \binom{0}{0}$, $F \cdot (1) = \binom{3}{0}$, $F \cdot (2) = \binom{6}{0}$

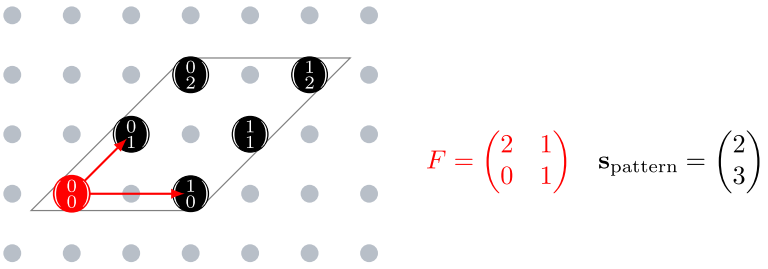

Fig. 4 Complex tile (sparse on a dimension and diagonal on the second). This last example illustrates how the tile can be sparse, thanks to the $\binom{2}{0}$ fitting vector, and non parallel to the axes of the array, thanks to the $\binom{1}{1}$ fitting vector

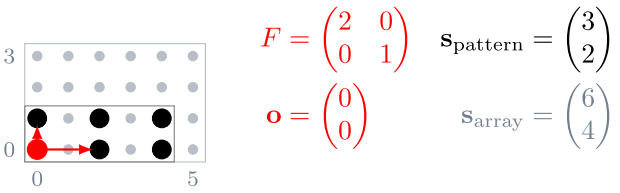

Fig. 5 A sparse tile aligned on the axes of the array

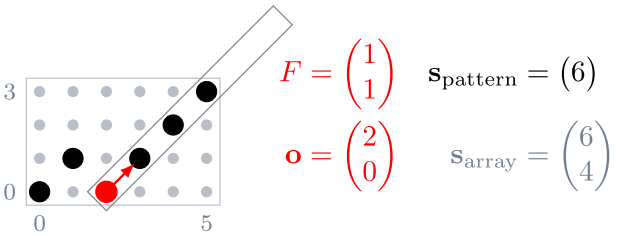

Fig. 6 The use of modulo. The pattern is here mono-dimensional, the fitting builds a diagonal tile that wraps around the array because of the modulo

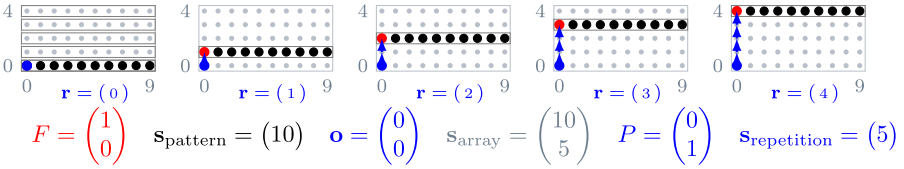

Fig. 7 Row paving. This figure represents the tiles for all the repetitions in the repetition space, indexed by $\mathbf{r}$. The paving vectors drawn from the origin $\mathbf{o}$ indicate how the coordinates of the reference element $\mathbf{ref}_{\mathbf{r}}$ of the current tile are computed. Here the array is tiled row by row

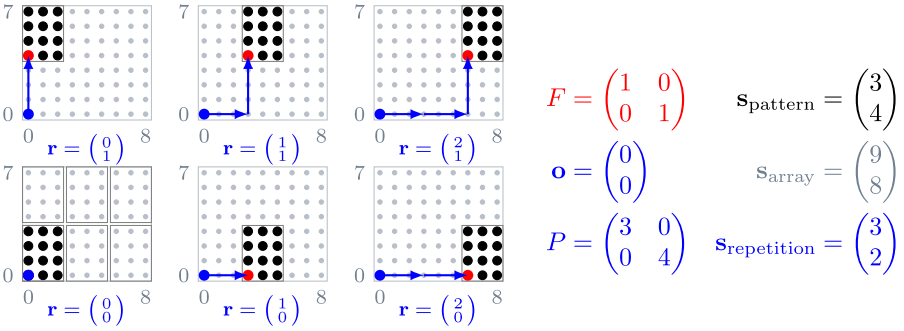

Fig. 8 A 2D pattern tiling exactly a 2D array

3.3.4 Summary

We can summarize all these explanations with one formula. For a given repetition index $\mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}$ and a given index $\mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}$ in the pattern, the corresponding element in the array has the coordinates

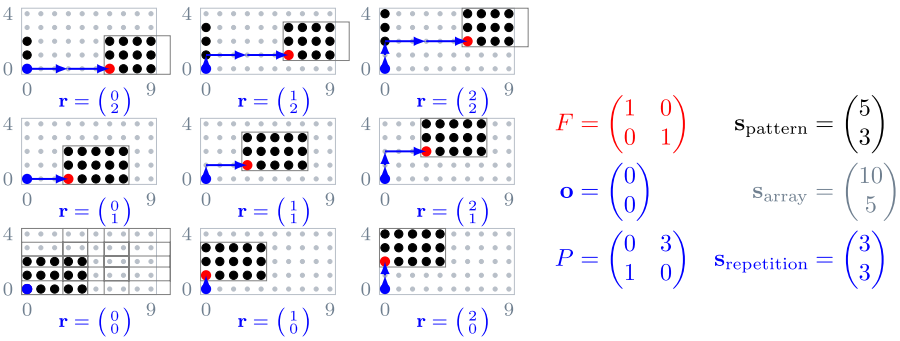

Fig. 9 The tiles can overlap and the array is toroidal

$$\mathbf{o} + (P \; F) \cdot \begin{matrix} \mathbf{r} \\ \mathbf{i} \end{matrix} \; mod \mathbf{s}_{\text{array}}, \tag{3}$$

where $\mathbf{s}_{\text{array}}$ is the shape of the array, $\mathbf{s}_{\text{pattern}}$ is the shape of the pattern, $\mathbf{s}_{\text{repetition}}$ is the shape of the repetition space, $\mathbf{o}$ is the coordinates of the reference element of the reference pattern, also called the origin, P is the paving matrix whose column vectors, called the paving vectors, represent the regular spacing between the patterns, F is the fitting matrix whose column vectors, called the fitting vectors, represent the regular spacing between the elements of a pattern in the array.

3.3.5 Linking the inputs to the outputs by the repetition space

The previous formulas explain which element of an input or output array one repetition consumes or produces. The link between the inputs and outputs is made by the repetition index, $\mathbf{r}$. For a given repetition, the output patterns (of index $\mathbf{r}$) are produced by the repeated task from the input patterns (of index $\mathbf{r}$). These pattern elements correspond to array elements through the tiles associated to the patterns. Thus the set of tilers and the shapes of the patterns and repetition space define the dependences between the elements of the output arrays and the elements of the input arrays of a repetition. As stated before, no execution order is implied by these dependences between the repetitions (Fig. 9).

To illustrate this link between the inputs and the outputs, we show in Fig. 3 several repetitions of the horizontal filter repetition. In order to simplify the figure and as the treatment is made frame by frame, only the first two dimensions are represented.[3] The sizes of the arrays have also been reduced by a factor of 60 in each dimension for readability reasons.

As we have just seen, Array-OL is indeed able to deal with any number of dimensions, sliding windows, different production and consumption rates, cyclic array dimensions and non parallel to the axes data access patterns. Considering that it is a hierarchical language, it only lacks a way to express delays and states.

[3] Indeed, the third dimension of the input and output arrays is infinite, the third dimension of the repetition space is also infinite, the tiles do not cross this dimension and the only paving vector having a non null third element is $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ along the infinite repetition space dimension.

4 Delays

4.1 Modeling delays

As shown earlier, in order to express the information needed for a complete specification of data-flow algorithms, languages like Array-OL are based on special formalisms. One of the common aspects in most of those languages is the component-based approach. The application is seen as a set of tasks that are connected and communicate through arrays. To maintain the data-flow semantics, no cycles are allowed in the graph representing the connections between tasks. Together with the single assignment constraint, that makes it impossible for tasks to maintain state information.

In this context, the concept of delay available in SDF and its extensions is of great importance to a data-flow language by allowing the construction of self-loops (states in SDF) into an application. A complete data-flow language must be able to express such self-loop constructions. For Array-OL, being a more expressive language than SDF and its multidimensional extensions, the delay concept is in consequence more complex. In SDF, a delay is an initial token or sample on the arc. In MDSDF delays are also multidimensional tuples and represent initial rows and columns. This means that the production of data is offset by the corresponding values while the consumption of data is not offset. This allows a special construction named *state* where an actor consumes tokens produced by itself but at a previous iteration. Rather than expressing initial tokens available on a connection, we have chosen to express uniform dependences between repetitions of the same component that we call inter-repetition dependences (Fig. 10).

4.2 Introductory example

To be able to represent loops containing inter-repetition dependences, we add the possibility to model uniform dependences between tiles produced and tiles consumed by a repeated component. A graphical representation can be seen in Fig. 11.

Formally an inter-repetition dependence connects an output port (p_{out} in the figure) of a repeated component with one of its input ports (p_{in} in the figure). *The shape of the connected ports must be identical*. The dependence connector is tagged with a dependence vector $\mathbf{d}$ that defines the dependence distance between the dependent repetitions. This dependence is uniform, which means identical for all the repetitions. When the source of a dependence is outside the repetition space, a default value is used. This default value is defined by a default connector connected to the same input port p_{in}.

When saying that a repetition $\mathbf{r}$ depends on another $\mathbf{r}_{\text{dep}}$ it means that at the execution time repetition $\mathbf{r}$ will receive as input on port p_{in} values produced by $\mathbf{r}_{\text{dep}}$ on its output port p_{out}. When a repetition takes values from a default connector it means that it will receive as input on port p_{in} values from a port connected with a default connector with p_{in} (or a tile of such a port if the default connector has an associated tiler as we will see later).

For the simple discrete integration example shown in Fig. 11 the patterns (and also the tiles) are single points. The uniform dependence vector $\mathbf{d} = (1)$ tells that each repetition $\mathbf{r}$ depends on repetition $\mathbf{r}_{\text{dep}} = \mathbf{r} - \mathbf{d} = \mathbf{r} - (1)$. In this case, the inter-repetition dependence is used to express that the output value of a repetition is used as input by the next repetition. Each repetition will take as input two values on its two ports, an input value from the tile and the result of the previous repetition. These values are added and provided on the output port which will serve as an input for the next repetition, and so on.

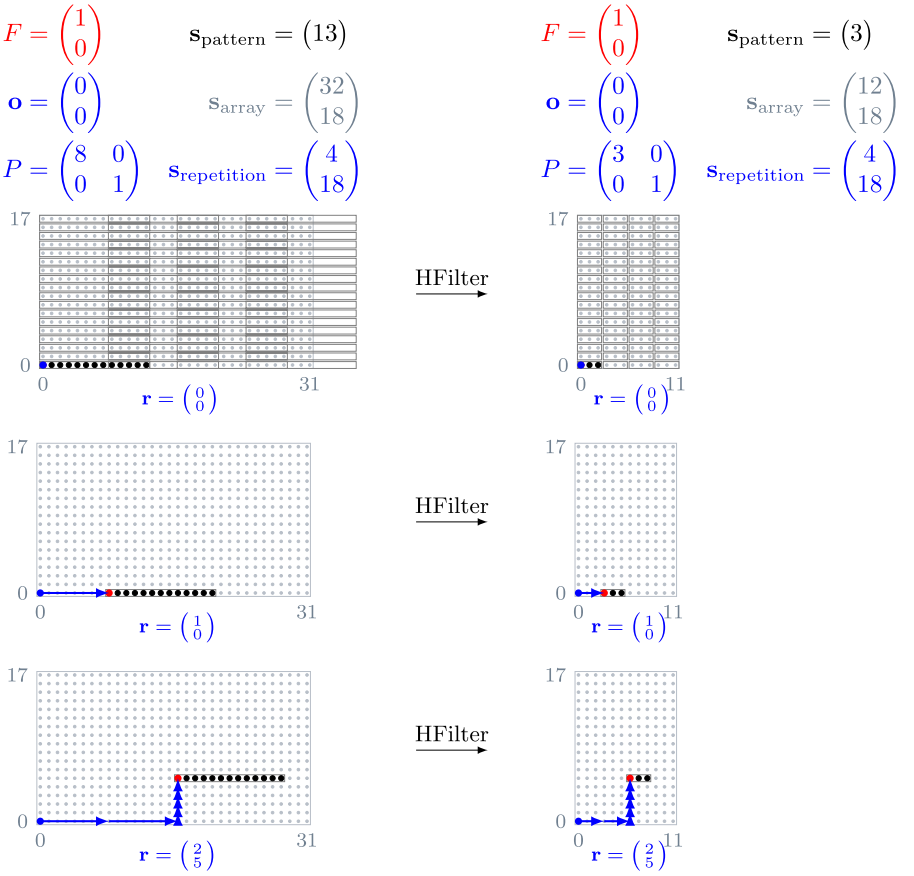

Fig. 10 Linking the input and outputs of the horizontal filter

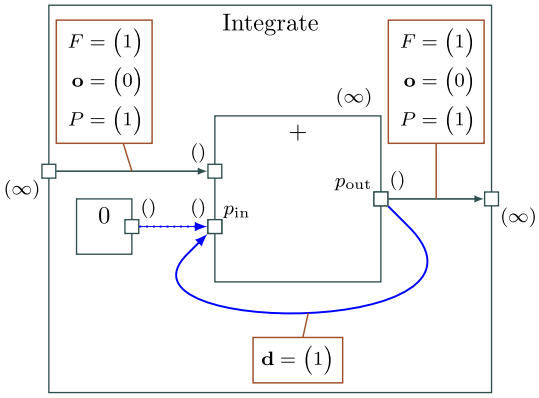

Fig. 11 A simple inter-repetition dependence. The self-loop represents the dependence inter-repetition, while the dotted connector represents the default connector

To start the computation, a default value of 0 is taken for repetition $\mathbf{r} = 0$ as indicated by the default connector. The computations are here sequentialized by the inter-repetition dependence.

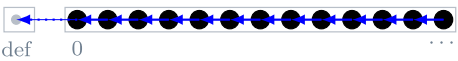

Fig. 12 In the mono-dimensional repetition space, the $\mathbf{d} = (1)$ inter-repetition dependence expresses that each repetition depends on its precedent. For the first one, the depending repetition is outside the repetition space, so it will use the default value as input

The inter-repetition dependence presented earlier is just a simple example used to illustrate the concepts of uniform dependence and default connector. The Array-OL with delays language allows the construction of much more complex structures, like multiple inter-repetition dependences or stride dependences. By combining them through the hierarchical structure of an application we can achieve complex dependences as we will see later on (Fig. 12).

4.3 Inter-repetition dependence definition

A complete inter-repetition dependence that, in the context of Array-OL with delays, allows the construction of complex dependences through hierarchy levels consists of the two elements used in the previous example (the uniform dependence vector and the default connector) to which we add some observations. First, the default connector can be connected to a port of the component containing the inter-repetition dependence. In this way we can establish connections between dependences found on different hierarchy levels. Second, the default connector can connect ports with different shapes, in which case we need a tiler on the connector to express the exact element-to-element relations. The tiler on the default connector allows having multiple default connectors to an inter-repetition dependence–from which only one is valid for each repetition needing a default value. A simple example for the applicability of such a construction is a two dimensional repetition space for which repetitions can have different default values in function of the direction in which we exit the repetition space (north, south, east or west).

4.3.1 Definition (inter-repetition dependence)

The formal specification of a complete inter-repetition dependence consists of:

- a repeated component c within a $\mathbf{s}_{\text{repetition}}$ repetition space,
- an inter-repetition dependence dep with the dependence vector $\mathbf{d}$; dep connects an output port p_{out} to an input port p_{in} (p_{out} and p_{in} have the same shape $\mathbf{s}$ and both belong to c),
- a set of n default connectors def_i ($0 \leq i < n$) connecting p_{in} to an output port p_i ($0 \leq i < n$) of other components,
- each default connector def_i has an associated tiler T_i, except the last one that may be lacking a tiler (in which case p_{n-1} must have the same shape as p_{in}); t represents the number of default connectors tagged with a tiler ($n - 1 \leq t \leq n$)

When computing the dependences, we have:

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \quad \mathbf{r}_{\text{dep}} = \mathbf{r} - \mathbf{d} \tag{4}$$

and if the dependent repetition is inside the repetition space ($\mathbf{0} \leq \mathbf{r}_{\mathbf{dep}} < \mathbf{s}_{\text{repetition}}$) then the repetition $\mathbf{r}$ depends on $\mathbf{r}_{\text{dep}}$ (the values produced by repetition $\mathbf{r}_{\text{dep}}$ on port p_{out} are consumed by repetition $\mathbf{r}$ on port p_{in}); otherwise repetition $\mathbf{r}$ takes its inputs from one of the default connectors.

In this case and when having more than one default connectors for an inter-repetition dependence, a selection between the available default connectors must be done. This selection is done in function of the repetition index $\mathbf{r}$ and the tilers associated to the default connectors. For each tiler a reference element will be computed, in the same way as for a normal tiler but without the use of modulo:

$$\forall i, 0 \leq i < t, \quad \mathbf{ref}_i = \mathbf{o}_i + P_i \cdot \mathbf{r}, \tag{5}$$

where $\mathbf{o}_i$ and P_i are the origin and the paving of the tiler T_i.

4.3.2 Validity property

The specification of the tilers of the default connectors must be done in such a way that for all the repetitions that need inputs from the default connectors, at most one of the computed references $\mathbf{ref}_i$ $(0 \leq i < t)$ is inside the shape of its corresponding port p_i. This valid reference $\mathbf{ref}_v$ thus verifies that $\mathbf{0} \leq \mathbf{ref}_v < \mathbf{s}_v$, where $\mathbf{s}_v$ is the shape of the port p_v. This reference together with the corresponding tiler T_v will be used to compute the tile to be passed to the input port p_{in} of the repetition $\mathbf{r}$ as the set of indices $\mathbf{e_i}$ verifying

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}, \mathbf{e_i} = \mathbf{ref}_v + F_v \cdot \mathbf{i} \mod \mathbf{s}_v \tag{6}$$

where $\mathbf{s}$ is the shape of p_{in} and $\mathbf{s}_v$ is the shape of p_v.

If none of the computed references is valid then the default connector not tagged with a tiler will be chosen. The exclusion between the tilers can be easily verified with the help of polyhedral algebra.

The tiler construction for the default connectors in order to allow only a valid one for each repetition might seem complicated but actually in real applications we do not need to use complicated default connectors. In most cases, the exclusion between the different default connectors is done by using the same tiler and just shifting the origins. An example and more details of such a construction are available in the next section in Fig. 17.

4.4 Multi-dimensional example

To better illustrate the concepts introduced in Array-OL with delays we present a more complex multi-dimensional example. This example also allows us to show how the inter-repetition dependences act when refactoring based on Array-OL transformations intervenes.

It is out of the scope of this paper to present these transformations, the whole set of transformations (fusion, tiling, change paving, collapse) and their implementation are described in Dumont (2005) and Soula et al. (2001). A great care has been taken in these transformations to ensure that they do not modify the semantics of the specifications. They only change the way the dependences are expressed in different hierarchical levels but not the precise element to element dependences. What is more important to this paper is to show that the semantics of inter-repetition dependences in Array-OL with delays are capable of expressing complex dependences like the ones computed after a refactoring stage (the dependences before and after such transformations must express the same exact dependences between repetitions in order not to modify the semantics of the application).

Inter-dependences interact with the transformations by the propagation of inter-repetition dependences through the part of the application involved in the transformation (this may mean inter-repetition dependence propagation, hierarchy transfer, modification but also dependence fusion).

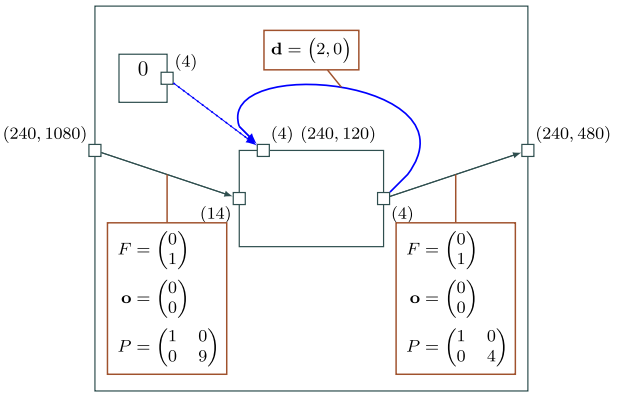

Fig. 13 A subset of Downscaler application, just the vertical filter for a single frame. An inter-repetition dependence has been added on the first dimension of the repetition space with a stride of 2

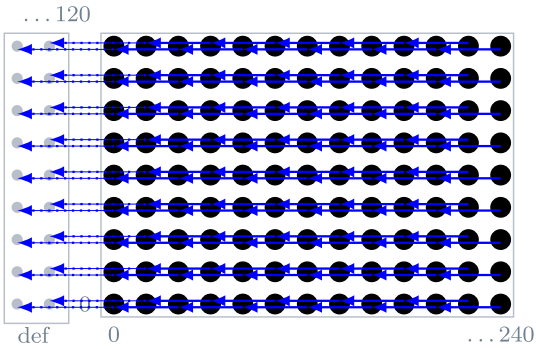

Fig. 14 Inter-repetition dependences in the 2D repetition space of our example

The application chosen is a subset of the downscaler application: just a repetition on which we have added an inter-repetition dependence on the first dimension of the repetition space with a stride of 2 (Fig. 13). This causes each repetition of the elementary filter task to have an additional input, the result from another repetition of the same component. The dependence vector $(2, 0)$ specifies horizontal dependences like shown in Fig. 14. The first two vertical columns of the repetition space have dependences outside the repetition space, therefore they will use the default values.

We can see in the Fig. 15 the same application after a *tiling* transformation for splitting the repetition space into blocks which has the effect of introducing a hierarchy level and splitting the repetition space between the two levels. After the transformation, the initial dependence must be modified to be adapted to the new application structure. It is split between the two hierarchy levels, remaining the same as before inside a block. In complement, in order to keep the dependences between repetitions from different blocks, a dependence between blocks on the higher hierarchy level was introduced. The dependence between elements found inside different blocks is done by connecting the two dependences with a default connector tagged with a tiler which has the role of making the correct correspondences between depending repetitions found in different blocks. This tiler is similar to the output tiler of the lowest hierarchy level but with a different origin computed in function of the dependence as a difference between the reference of the depending block and the reference of the dependence

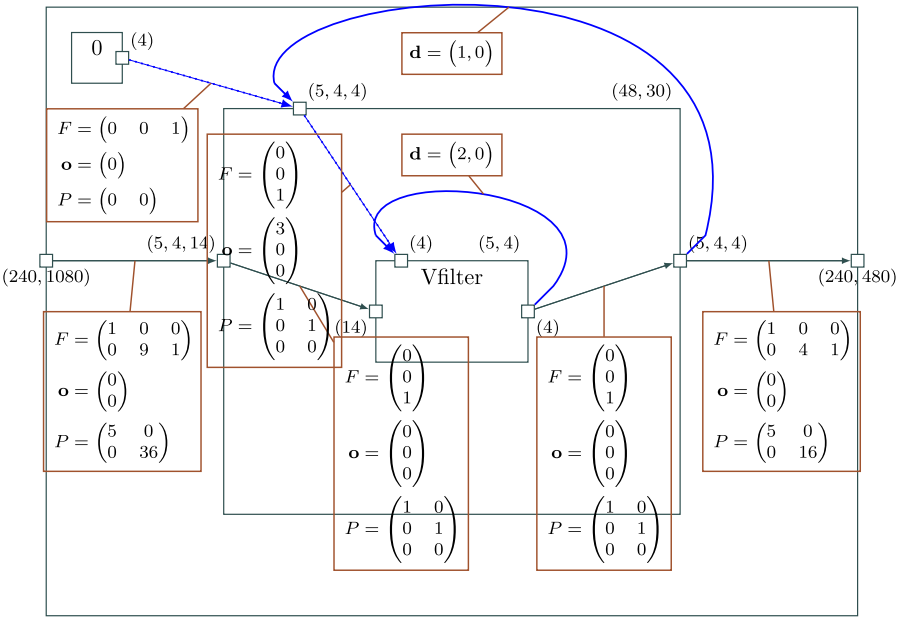

Fig. 15 The same application after a *tiling* transformation for splitting the repetition space into (5, 4)-blocks

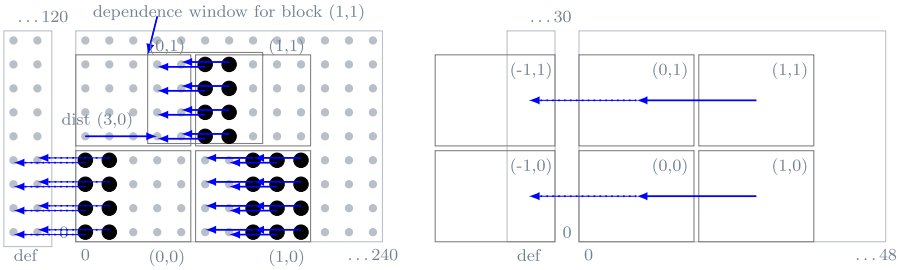

Fig. 16 Inter-repetition dependences after block splitting. We show on the left only parts of the repetition space: In block (1, 0) we have all the repetitions that depend on repetitions inside the same block, in block (1, 1) repetitions that depend on repetitions found in another block and in block (0, 0) repetitions that will use the default value. On the right, we show the inter-repetition dependences between the blocks

window—the dependence window for a block is a block of the same dimensions that contains all the depending repetitions for this block (like shown in Fig. 16). The similarity between the tilers is obvious when analyzing in more details the structure: the two tilers must have the same array dimensions (caused by being connected by the higher level inter-repetition dependence), the same pattern dimension (connected by the lower level inter-repetition dependence) and also having the same repetition space.

As an observation for the dependence between blocks, blocks (0, 1) and (0, 0) depend on blocks outside the repetition space, therefore they will use blocks filled with default values which are copies of the default value found on the initial default connector.[4] The choice of

[4] They are just virtual copies at specification time, in order to reduce the memory requirements at execution no real copies are needed.

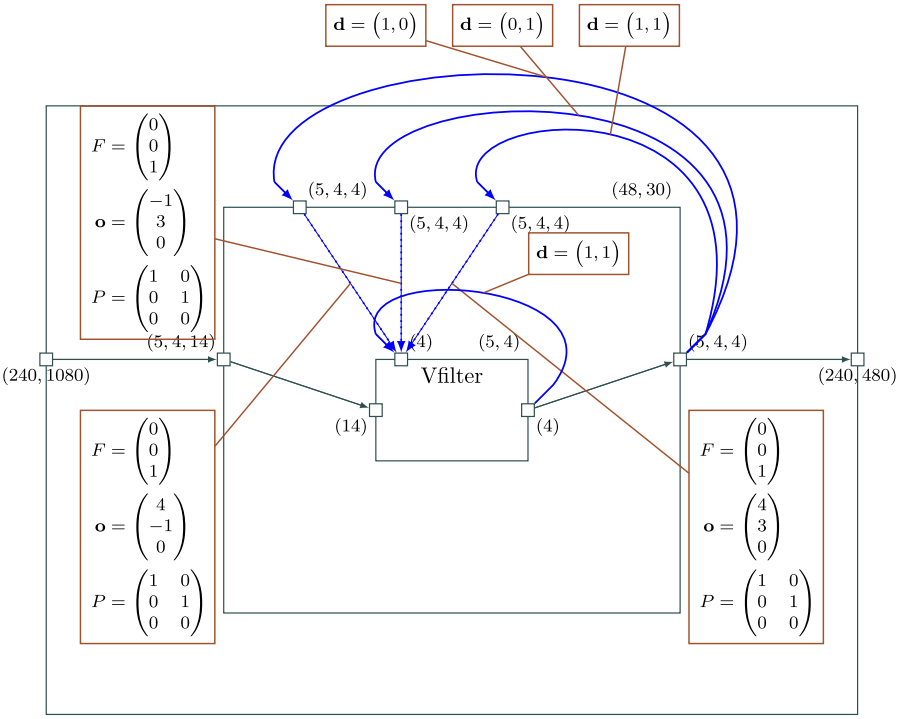

Fig. 17 The same application, the same transformation, the only difference is the dependence vector which is now a diagonal one. We have chosen not to over-charge the figure with redundant information that are the same as in Fig. 15 (tilers, default connector); changes appear only for the inter-repetition dependences

using such blocks is due to the constraint to have ports with identical shapes at the two ends of an inter-repetition dependence. The copy of the initial default values is expressed by the associated tiler which build a pattern larger than the array by replication. The zeros from the paving and fitting in the tiler, except for the last column of the fitting, express that the array of dimension (4) corresponds to the last dimension of the pattern, while on the other dimension of the fitting and paving, this array will be duplicated.

More complex inter-dependence through the hierarchy structures can be imagined, a simple variation in the inter-dependence vector in our example could cause the repetitions from a block to depend on repetitions from more that one block. This means that we need more than one inter-dependence between blocks and for each block dependence a connection to the lower level dependence. This case is presented in Fig. 17, where we have the result of the same transformation as before applied to the same application except for the initial dependence vector which is now a diagonal one of (1, 1). The different dependence vector has an impact only on the inter-dependence repetitions elements and in this way we have a clear separation between the transformations and the inter-repetition dependences (Fig. 18).[5]

The dependence window of block (1, 1) intersects three other neighbor blocks (Fig. 19) and therefore we have three dependences between blocks, and three default connectors at the lowest hierarchy level in order to make the selection of the appropriate dependence block.

[5] This is extremely important because they do not interfere with the automatic transformations engine, we need just a post-refactoring stage for an automatic inter-repetition dependence transformation to the new application structure.

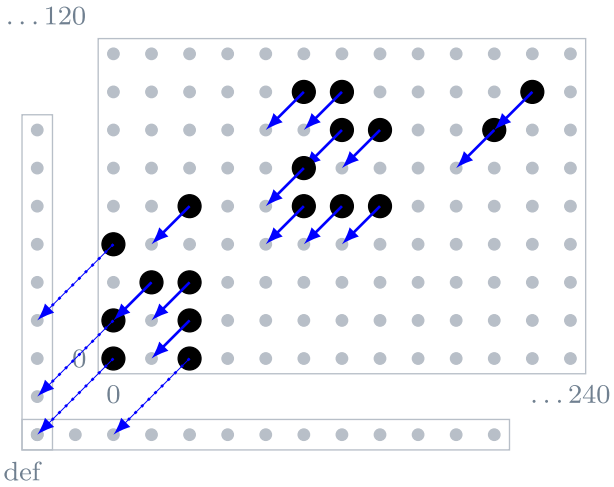

Fig. 18 Dependences between repetitions for the diagonal inter-repetition dependence. Just some random repetitions are shown in the figure

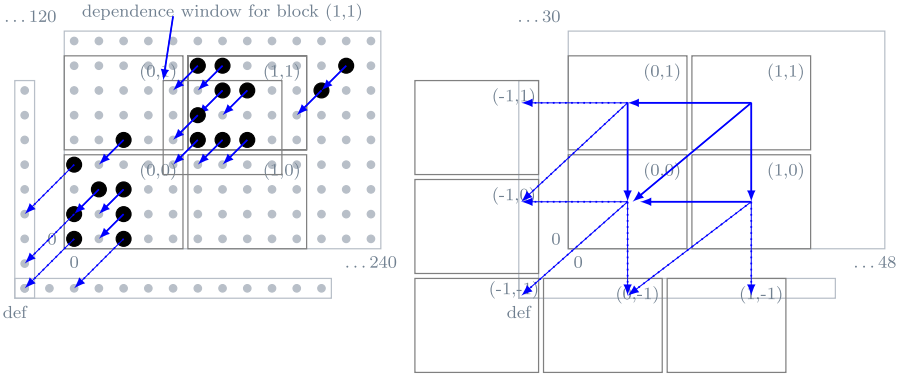

Fig. 19 Same diagonal dependences after block splitting on the left. Triple dependence between blocks on the right

The selection is done by the tilers on the default connectors, tilers being similar to each other and to the output tiler as before, with the exception of the origins. The different origins ensure the exclusive selection of the default connectors (the origin represent the difference between the reference of the depending block and the reference of the dependence window). As the absolute of the difference between any two origins exceeds, on at least one dimension, the size of the array (the same array shape for all the tilers) and the same paving matrix for the three tilers guaranties that for a repetition just one tiler will have a valid reference.

4.4.1 Proof of tiler exclusion

Presuming that for a repetition **r** having the dependence repetition outside the repetition space we have one tiler (T_v) from the default connectors that produces a reference element inside the array dimensions.

$$\mathbf{ref}_v = \mathbf{o}_v + P_v \cdot \mathbf{r} \tag{7}$$

and

$$\mathbf{0} \leq \mathbf{ref}_v < \mathbf{s}. \tag{8}$$

For any other tiler T_i we have

$$\forall i, 0 \leq i < t, i \neq v, \mathbf{ref}_i = \mathbf{o}_i + P_i \cdot \mathbf{r} \tag{9}$$

and we have identical paving matrices and array dimensions, while the difference between the two origins is δ_i:

$$P_i = P_v, \mathbf{o}_i = \mathbf{o}_v + \delta_i. \tag{10}$$

Replacing into Eq. 9 we have

$$\mathbf{ref}_i = \mathbf{o}_v + \delta_i + P_v \cdot \mathbf{r} = \delta_i + \mathbf{ref}_v. \tag{11}$$

As by hypothesis $|\delta_i| \not< \mathbf{s}$, either $\mathbf{ref}_i \not\geq \mathbf{0}$ or $\mathbf{ref}_i \not< \mathbf{s}$, which means that tiler i is not valid. □

4.5 Discussion

The Array-OL specification language allows much more complex constructions than SDF and its multidimensional extensions. The only restriction was expressing delays but with the inter-repetition dependence extension we can say that Array-OL has become a language capable of expressing not only delays or states but much more complex dependences. All this makes in our opinion Array-OL with delays a "complete" multi-dimensional intensive signal processing specification language.

Furthermore, the inter-repetition dependences are capable of expressing complex dependences connected through the hierarchy levels, extremely important to Array-OL in the context of Array-OL transformations, key tools when talking about refactoring, optimizations and scheduling with Array-OL (Amar et al. 2005). These transformations were extensively studied and their functionality was formalized, proved and implemented into a refactoring tool using a formalism based on linear algebra designed specially for Array-OL.[6] A comparative study between these transformations and the loop transformations in the context of program optimizations can be found in Glitia and Boulet (2008). As already said, these transformations are not influenced by the inter-repetition dependences. The transformations engine acts on redistributing repetitions through the application hierarchy and it guaranties not to modify the semantics of the application. Inter-repetition dependences, like shown in the examples, are capable of expressing dependences after such transformations also without modifying the semantics of the application.

By combining inter-repetition dependences with the hierarchical structure of Array-OL we can construct non-uniform dependences. An example of such construction could be a two-dimensional repetition space with a linear dependence on each row (each element depends on the precedent) while the first element depends on the last element of the previous row to express a Z-shaped dependence chain.

[6] ODT (Opérateurs de Description de Tableau in French)—Array Description Operators in English—more details in bibliography (Soula 2001; Dumont 2005).

Inter-repetition dependences are modeled in Array-OL using the same concepts available in Array-OL, like repetition space, tilers (paving, fitting, origin) etc. and in this way there was no need to add too many new concepts.

Another very important property of SDF and its extensions is the possibility to statically schedule the applications they describe. As said before, there exists a restriction of Array-OL that is schedulable statically as explained in Boulet (2008). Most reasonable Array-OL applications are actually in that restriction. Scheduling loop nests with uniform dependences is well known since the work of Darte and Robert (1994). Using these two results together it is thus possible to schedule statically most applications specified with Array-OL with delays.

5 Conclusion

We have presented in this paper the Array-OL multidimensional specification model and compared it to the other models designed for intensive signal processing applications such as video processing and detection systems. In order to allow the construction of structures like delays and states, the concept of inter-repetition dependence was introduced in Array-OL. The formalism of this concept based on the same structures as Array-OL is presented in the paper. The resulting specification model, Array-OL with delays, is thus able to express multidimensional signal processing applications with the common patterns of this application domain: sliding windows, over- and sub-sampling, cyclic array dimensions, states, delays and hierarchy while providing an expression of the full potential parallelism of the applications thanks to its dedicated langage-level features.

As a complement, inter-repetition dependences can be used not only to model applications with delays and states but also for specifying repetitive architectures and control structures based on mode automata. Indeed, the Array-OL with delays concepts are used in the MARTE (Modeling and Analysis of Real-Time and Embedded systems) standard UML profile (ProMarte partners 2008) (Repetitive Structure Modeling chapter) not only to model applications but also hardware architectures and the distribution of applications on architectures (Array-OL concepts can be extended to express the mapping of repetitive applications on repetitive architectures) and also to express control structures into applications.

When talking about architectures and inter-repetition dependences there are some slight differences. Array-OL is used to model repetitive architectures, like a grid of processors. In such cases, to express the connections between different processors in the grid, we can use inter-repetition dependences. To be able to express cycle connections in an architecture model, an inter-repetition dependence can be tagged as cyclic, in which case the dependence repetition is calculated using modulo and there is no need for default connectors.

The concept of inter-repetition dependence is also a base component when modeling control. In Labbani et al. (2006, 2005) control is expressed by mode automata using inter-repetition dependences.

Everything presented in this paper (Array-OL with delays, refactoring transformations, hardware architecture and distribution modeling, control with mode automata) is implemented in the model-driven engineering framework Gaspard2 to codesign intensive signal processing applications on systems-on-chip. The specification language of Gaspard2 is a subset of MARTE based on Array-OL with delays from which can be derived cosimulation code in SystemC (Piel et al. 2008), verification code in synchronous languages (Yu et al. 2008; Gamatié et al. 2008), or synthesis code in VHDL Le Beux et al. (2007); Sébastien Le Beux et al. (2008).

References

Amar, A., Boulet, P., & Dumont, P. (2005). Projection of the Array-OL specification language onto the Kahn process network computation model. In *International symposium on parallel architectures, algorithms, and networks*, Las Vegas, Nevada, USA.

Attali, I., Caromel, D., Syau Chen, Y., Luc Gaudiot, J., & Wendelborn, A. L. (1995). A formal semantics for sisal arrays. In *Proceedings of joint conference on information service*.

Bilsen, G., Engels, M., Lauwereins, R., & Peperstraete, J. (1995). Cyclo-static data flow. In *International conference on acoustics, speech, and signal processing*, Detroit, MI, USA.

Boulet, P. (2008). *Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing*. Research Report RR-6467, INRIA.

Chen, M. J., & Lee, E. A. (1995). Design and implementation of a multidimensional synchronous dataflow environment. In *1995 Proceedings of IEEE Asilomar Conference on Signal, Systems, and Computers*.

DaRT Team LIFL/INRIA, Lille, France. (2008). Graphical array specification for parallel and distributed computing (GASPARD2). https://gforge.inria.fr/projects/gaspard2/.

Darte, A., & Robert, Y. (1994). Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel Distributed Systems, 5*(8), 814–822.

de Dinechin, F., Quinton, P., & Risset, T. (1995). Structuration of the alpha language. In *Programming models for massively parallel computers* (pp. 18–24). Berlin, Germany.

Demeure, A., & Del Gallo, Y. (1998). An array approach for signal processing design. In *Sophia-Antipolis conference on micro-electronics (SAME 98)*, France.

Demeure, A., Lafarge, A., Boutillon, E., Rozzonelli, D., Dufourd, J.-C., & Marro, J.-L. (1995). Array-OL: Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France (In French).

Dumont, P. (2005). *Spécification Multidimensionnelle pour le traitement du signal systématique*. Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille.

Dumont, P., & Boulet, P. (2005). *Another multidimensional synchronous dataflow: Simulating Array-OL in ptolemy II*. Research Report RR-5516, INRIA.

Gamatié, A., Rutten, E., Yu, H., Boulet, P., & Dekeyser, J.-L. (2008). Synchronous modeling and analysis of data intensive aplications. *EURASIP Journal on Embedded Systems*, Article ID 561863, p. 22.

Gaudiot, J.-L., DeBoni, T., Feo, J., Böhm, W., Najjar, & W., Miller, P. (2001). *Compiler optimizations for scalable parallel systems: Languages, compilation techniques, and run time systems*. The Sisal project: Real world functional programming (pp. 45–72). NY, USA: Springer-Verlag.

Glitia, C. & Boulet, P. (2008). High level loop transformations for multidimensional signal processing embedded applications. In *SAMOS 2008 workshop*, Samos, Greece.

Joachim Keinert, J. T., & Haubelt, C. (2005). *Windowed synchronous data flow*. Technical report co-design-report 02, 2005. Department of Computer Science 12, Hardware-Software-Co-Design, University of Erlangen-Nuremberg, Am Weichselgarten 3, D-91058 Erlangen, Germany.

Karp, R. M., Miller, R. E., & Winograd, S. (1967). The organization of computations for uniform recurrence equations. *Journal of ACM, 14*(3), 563–590.

Keinert, J., Haubelt, C., & Teich, J. (2006). Modeling and analysis of windowed synchronous algorithms. In *International conference on acoustics, speech and signal processing (ICASSP)* (pp. III-892– III-895).

Labbani, O., Dekeyser, J.-L., Boulet, P., & Éric Rutten. (2005). Introducing control in the Gaspard2 data-parallel metamodel: Synchronous approach. *International workshop MARTES: Modeling and analysis of real-time and embedded systems (in conjunction with 8th international conference on model driven engineering languages and systems, MoDELS/UML 2005)*.

Labbani, O., Dekeyser, J.-L., Boulet, P., & Rutten, E. (2006). UML2 profile for modeling controlled data parallel applications. In *FDL'06: Forum on specification and design languages*, Darmstadt, Germany.

Le Beux, S., Marquet, P., & Dekeyser, J.-L. (2007). A design flow to map parallel applications onto FPGAs. In *17th IEEE international conference on field programmable logic and applications, FPL*, Amsterdam, Netherlands.

Le Verge, H., Mauras, C., & Quinton, P. (1991). The alpha language and its use for the design of systolic arrays. *The Journal of VLSI Signal Processing, 3*(3), 173–182.

Lee, E. A. (1993). 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, North-Holland.

Lee, E. A., & Messerschmitt, D. G. (1987a). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*.

Lee, E. A., & Messerschmitt, D. G. (1987b). Synchronous data flow. *Proceedings of the IEEE, 75*(9), 1235–1245.

Murthy, P., & Lee, E. A. (1995). *A generalization of multidimensional synchronous dataflow to arbitrary sampling lattices*. Technical report UCB/ERL M95/59, EECS Department, University of California, Berkeley.

Murthy, P. K. (1996). *Scheduling techniques for synchronous and multidimensional synchronous dataflow*. PhD thesis, University of California: Berkeley, CA.

Murthy, P. K., & Lee, E. A. (2002). Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing, 50*(8), 2064-2079.

Piel, E., Attitalah, R. B., Marquet, P., Meftali, S., Niar, S., Etien, A., Dekeyser, J.-L., & Boulet, P. (2008). Gaspard2: From MARTE to SystemC simulation. In *Modeling and analyzis of real-time and embedded systems with the MARTE UML profile DATE'08 workshop*.

ProMarte partners. (2008). UML profile for MARTE, Beta 2. http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf.

Le Beux, S., Marquet, P., & Dekeyser, J-.L. (2008). A design space exploration flow for fpga implementation of intensive signal processing applications. In *Conference on design and architectures for signal and image processing (DASIP 2008)*, Bruxelles, Belgium.

Scholz, S.-B. (2003). Single assignment c: Efficient support for high-level array operations in a functional setting. *J. Funct. Program., 13*(6), 1005–1059.

Soula, J. (2001). *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 (In French).

Soula, J., Marquet, P., Dekeyser, J.-L., & Demeure, A. (2001). Compilation principle of a specification language dedicated to signal processing. In *Sixth international conference on parallel computing technologies, PaCT 2001*, Novosibirsk, Russia. Lecture Notes in Computer Science (Vol. 2127, pp. 358–370).

Thies, W., Karczmarek, M., & Amarasinghe, S. (2002). StreamIt: A language for streaming applications. In *Compiler construction: 11th international conference, CC 2002, held as part of the joint European conferences on theory and practice of software, ETAPS 2002, Grenoble, France, April 8–12, 2002, Proceedings*. Volume of 2304/2002 of Lecture Notes in Computer Science (pp. 49–84). Springer Berlin: Heidelberg.

Yu, H., Gamatié, A., Éric Rutten, & Dekeyser, J.-L. (2008). *Embedded systems specification and design languages, selected papers from FDL 2007*. Model transformations from a data parallel formalism towards synchronous languages. Springer.

Author Biographies

Calin Glitia was born in Oradea, Romania, on May 5, 1982. In 2005 he obtained an engineer degree at ôPolitehnicaö University of Timisoara, Romania and in 2006 a master degree at Ecole Normale SupTrieure of Lyon, France. He is doing now his PhD in the DaRT INRIA team at University of Lille, France, under the direction of Pierre Boulet. The subject of his PhD is: "Optimizations for intensive systematic processing applications on System-on-Chips".

Philippe Dumont was born in Limoges, France, on January 9, 1978. He studied in the University of Limoges, Bordeaux and Lille. He has done his PhD under the direction of Pierre Boulet in the DaRT INRIA tean located at the University of Lille, France. The subject of his PhD was: "Multidimensionnal Specification for Systematic Signal Processing", he has defended this PhD in 2005. He works now for NXP semiconductors.

Pierre Boulet was born in Lille, France, on December 11, 1970. He earned a DEA d'Informatique Fondamentale in 1993 and a PhD in 1996 from the Ecole Normale Supérieure de Lyon, France. He is currently professor of computer science in the Université des Sciences et Technologies de Lille, France. His interests range from parallelism, compilation, embedded system co-design to model driven engineering and synchronous languages.