Interaction between inter-repetition dependences and high-level transformations in Array-OL

Calin Glitia and Pierre Boulet,
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
INRIA Lille - Nord Europe
59655 Villeneuve d'Ascq Cedex, FRANCE

Abstract

Systematic 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). The Array-OL specification language is designed to allow the direct manipulation of these different dimensions with a high level of abstraction. An extension of Array-OL was previously introduced in order to allow the modeling of uniform inter-repetition dependences. This article studies the interaction between these dependences and the high-level transformations designed to allow to adapt the application to the execution, already available on Array-OL.

1 Introduction

Computation intensive multidimensional applications are predominant in several application domains such as image and video processing or detection systems (radar, sonar).

The Array-OL (Array Oriented Language) specification language is designed to provide ways 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. Some 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 stateful computations such as recursive filters.

A complete formal specification of Array-OL is available in [2] and a comparison with several languages (or models of computation) dedicated to signal processing is available in [7]. Most of the compared languages are based on SDF (Synchronous Data Flow) [9] or on its multidimensional extension, MDSDF (Multi-dimensional Synchronous Data Flow) [10], like GMDSDF [10] or WSDF [8], which are the languages that share most common elements with Array-OL. A detailed comparison of MDSDF, GMDSDF and Array-OL (without delays) is available in [5].

Array-OL was able to deal with all the features mentioned earlier, with the exception of state structures. A recent extension in Array-OL was proposed in [7] to cope with expressing state structures, the inter-repetition dependence extension. Such dependence consists of a self-loop on a repetition task that expresses uniform dependences between the repetitions on that task. The Array-OL language expresses the minimal order of execution that leads to the correct computation. This is a design intension and lots of decisions can and have to be taken when mapping an Array-OL specification onto an execution platform. A set of Array-OL code transformations is available [4, 12], designed to allow to adapt the application to the execution, allowing to choose the granularity of the flows and a simple expression of the mapping by tagging each repetition by its execution mode: data-parallel or sequential. These transformations act on manipulating the hierarchical structure of an application and by distributing repetitions through this hierarchy, guaranteeing that the semantics of the application remain unchanged. With the extension of the language, these transformations should guarantee in

addition that the uniform dependences ramain also unchanged.

Our interest in this paper is exclusively the interaction between these dependences and the Array-OL transformations. It is essential in order to preserve the techniques of passing to an execution model for Array-OL models with dependences. We identifying the rules that express the relations between the transformations and the inter-repetition dependences. Based on these rules, we propose an algorithm for transforming the inter-repetition dependences parallel to the available Array-OL transformation engine.

We recall the bases of Array-OL in section 2, followed the presentation of the extension in section 3 and by a short presentation of the Array-OL transformations in section 4. A formal analysis of the interaction and based on this the algorithm allowing the transformation of the inter-repetition dependences is presented in section 5.

2 Array-OL - principles

The initial goal of Array-OL is to give a mixed graphical-textual language to express multidimensional intensive signal processing applications. 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.
- Array-OL is a *data dependence expression* language.
- It is a *single assignment* formalism.
- Data accesses are done through uniform sub-arrays, called *patterns*.
- The language is *hierarchical* to allow descriptions at different granularity levels and to handle the complexity of the applications.
- The spatial and temporal dimensions are treated equally in the arrays.
- The arrays are seen as tori. Indeed, some spatial dimensions may represent some physical tori, like hydrophones around a submarine.

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.

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.

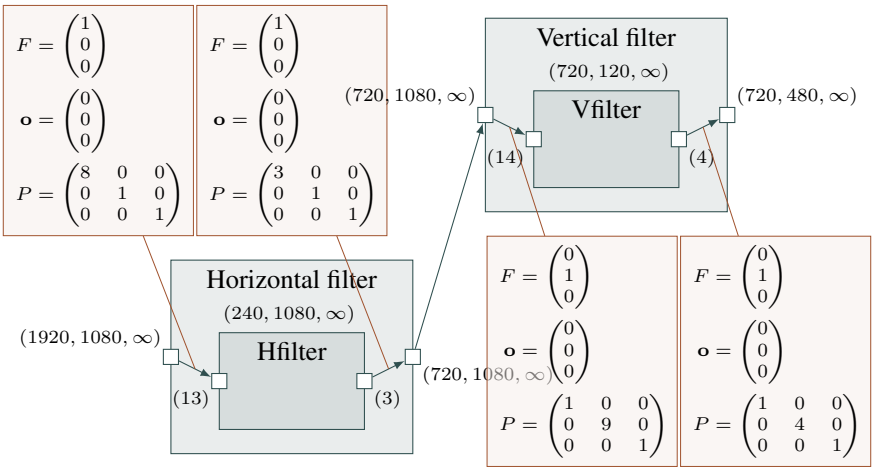
All the data exchanged between the tasks are arrays. These arrays are multidimensional and are characterized by their *shape*, the number of elements on each of their dimension. As said above, the Array-OL model is single assignment. One manipulates *values* and not *variables*. Time is thus represented as one (or several) dimension of the data arrays.

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. 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.

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:

- F: a *fitting* matrix whose column vectors represent the regular spacing between the elements of a pattern in the array.
- $\mathbf{o}$: the *origin* of the *reference pattern* (for the *reference repetition*).
- P: a *paving* matrix whose column vectors represent the regular spacing between the patterns.

We can summarize the pattern construction with one formula. For a given repetition index $\mathbf{r}, \mathbf{0} \leq$



Each of the filter has a repetitive functionality, described with the tilers. For example, the horizontal filter's elementary component takes a window of 13 elements that slides with 8 elements on each line of each image frame and produces 3 elements.

Figure 1. Example: downscaler from high definition TV to standard definition TV

$\mathbf{r} < \mathbf{s}_{\text{rep}}$ 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

$$\mathbf{o} + (P\ F) \cdot \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s}_{\text{array}}, \qquad (1)$$

where $\mathbf{s}_{\text{array}}$ is the shape of the array, $\mathbf{s}_{\text{pattern}}$ is the shape of the pattern, $\mathbf{s}_{\text{rep}}$ is the shape of the repetition space. The link between the inputs and outputs is made by the repetition index, $\mathbf{r}$. The representation of a Downscaler application from high definition TV to standard definition TV is presented in *Figure 1*.

A complete definition of a semantics for Array-OL language can be found in [2], together with a set of construction rules and the way to statically verify them that ensures that a specification admits a static schedule. An application is statically schedulable if the dependence relation between the calls to the elementary tasks is a strict partial order. One of the rules stipulates that no cycle in the graph of a compound task is allowed. This restriction forbids the construction of stateful structures. In order to overcome this language restriction, an extension of the Array-OL language was introduced in [7], which allows cycles on a repeated task represented as an uniform dependence, called *inter-repetition dependence*.

3 Modeling uniform dependences

Formally an inter-repetition dependence connects an output port of a repeated component with one of its input ports. 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. 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 values produced by $\mathbf{r}_{\text{dep}}$ on its output port. In *Figure 2*, the dependence vector $(1, 1)$ specifies diagonal dependences like shown in *Figure 3*.

Definition (inter-repetition dependence). The formal specification of a complete inter-repetition dependence consists of:

- a repeated component c within a $\mathbf{s}_{\text{rep}}$ 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,

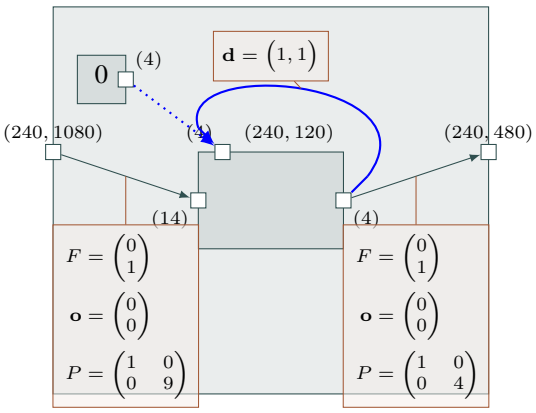


Figure 2. Dependence example

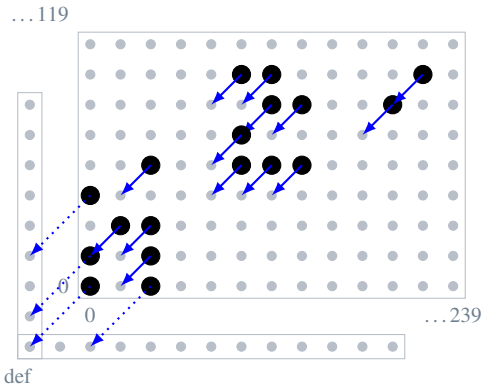


Figure 3. Diagonal dependence

- each default connector def_i has an associated tiler T_i, except the last one that may be lacking a tiler (in which case $p_{\text{n-1}}$ 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:

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{rep}}, \mathbf{r}_{\text{dep}} = \mathbf{r} - \mathbf{d} \qquad (2)$$

and if the dependent repetition is inside the repetition space ($\mathbf{0} \leq \mathbf{r_{dep}} < \mathbf{s}_{\text{rep}}$) 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.

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. The reference element will be computed in the same way as for a normal tiler but without the use of modulo:

$$\forall i, 0 \leq i < t, \mathbf{ref}_i = \mathbf{o}_i + P_i \cdot \mathbf{r}, \qquad (3)$$

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

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}, \mathbf{e_i} = \mathbf{ref}_v + F_v \cdot \mathbf{i} \bmod \mathbf{s}_v \qquad (4)$$

where $\mathbf{s}$ is the shape of p_{in} and $\mathbf{s}_v$ is the shape of p_v. The exclusion between the tilers can be easily verified with the help of polyhedral algebra.

4 Array-OL transformations

As mentioned, an Array-OL specification that respects the construction rules is statically schedulable. Any schedule that respects the strict partial order between the calls to the elementary tasks of an application will compute the same result without any deadlock. Is a design intension that, by expressing the minimal order of execution, lots of decisions can and have to be taken when mapping an Array-OL specification onto an execution platform: how to map the various repetition dimensions to time and space, how to place the arrays in memory, how to schedule parallel tasks on the same processing element, how to schedule the communications between the processing elements? Mapping compounds is not specially difficult. The problem comes when mapping repetitions. This problem is discussed in details in [1] where the authors study the projection of Array-OL onto Kahn process networks. A representative illustration of the problem is the presence of any intermediary array that contains an infinite dimension, which would cause the execution to be stalled in that point. The key point is that some repetitions can be transformed to flows. In that case, the execution of the repetitions is sequentialized (or pipelined) and the patterns are read and written as a flow of tokens (each token carrying a pattern). This can be achieved by refactoring the application using the Array-OL transformations. Using the hierarchy, we intend to isolate the infinite dimensions at the top hierarchical level of the application (which will represent the data-flow).

The Array-OL code transformations can be used to adapt the application to the execution, allowing to choose the granularity of the flows and a simple expression of the mapping by tagging each repetition by its execution mode: data-parallel or sequential. A great care has been taken with these transformations to ensure that they do not modify the precise element to element dependences [4, 12], by using a formalism based on linear algebra designed specially for Array-OL[1]. A comparative study between these transformations and the loop transformations in the context of program optimizations can be found in [6]. Although the similarities between the two types of transformations are obvious, the two are situated at completely different levels and their role is different. The loop transformations are at the level of execution and are used mainly in compiler optimizations, while the Array-OL transformations are situated at a high-level of specification and their role is to adapt the Array-OL specification to the execution model and platform. By refactoring the application we can eliminate deadlocks, reduce intermediary arrays or change the granularity of the application and also facilitate architecture exploration. Nontheless, the use of the two types of transformations is not exclusive, after using the Array-OL transformations at the specification level, the loop transformations can be used when compiling the generated code.

5 Dependences after transformation

Regardless of their role, all the Array-OL transformations have similar impact on an application, when talking about repetitions and hierarchy. Generalizing, they act on redistributing repetitions through the hierarchy levels, with the creation or suppression of hierarchy levels if needed. Furthermore, a transformation involves a maximum of two successive hierarchy levels.

Taking each transformation one by one, we have:

- *Fusion* takes one level of hierarchy, creates a superior hierarchy level for the computed common repetition, while what is left of the initial repetitions is placed on the inferior hierarchy level.

- *Change paving* (either by dimension creation or by linear growth) has no impact on the hierarchy levels, it just moves repetitions from the superior level to the inferior level of the hierarchy.

- *Tiling* splits a repetition into blocks, by creating a hierarchy level.

- *Collapse*, by being the opposite of fusion and tiling, suppresses the superior hierarchy level, its repetitions being added to each of the inferior level repetitions.

The Array-OL transformations guarantee that the semantics of the application are not modified. This implies that the repetitions stay the same after the transformation, they are just rearranged through the hierarchy and this forces a rearrangement of the eventual inter-repetition dependences.

The issue can be formulated as follows: *Having the structure before and after the transformation (represented both times by a one or two-level hierarchy of repetitions) together with an inter-repetition dependence before the transformation, the inter-repetition dependence(s) that express the same exact dependences on the new transformed repetitions have to be computed.*

To do so, a connection between the initial and final repetitions in a transformation must be identified by manipulating the formalism behind Array-OL and some constrains that ensure that the semantics of the application do not change. As said, a transformation makes changes just through the repetitions involved in the transformations. The interface with the rest of the application must remain the same; the arrays that comunicate to the rest of the application and the way they are consumed/produced must remain unchanged. Through these arrays, connections between repetitions before and after a transformation can be identified.

We start by expressing the connections between the repetitions and the arrays for the two scenarios: one or two hierarchy levels of repetitions.

One level. The connection between the repetition $\mathbf{s}_{\text{rep}}$ and the array $\mathbf{s}_{\text{array}}$ in *Figure 4* is done through the tiler T. Using the rule for pattern construction (1), we have:

$$\forall\, \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{rep}}, \mathbf{ref}_{\mathbf{r}} = \mathbf{o} + P \cdot \mathbf{r} \bmod \mathbf{s}_{\text{array}} \quad (5)$$

Two levels. The connection between the two repetition spaces $\mathbf{s}_{\text{rep}_{\text{sup}}}$ and $\mathbf{s}_{\text{rep}_{\text{inf}}}$ and the array $\mathbf{s}_{\text{array}}$ (*Figure 5*) is done through the two tilers T_{sup} and

[1] ODT (*Opérateurs de Description de Tableau* in French) – *Array Description Operators* in English .

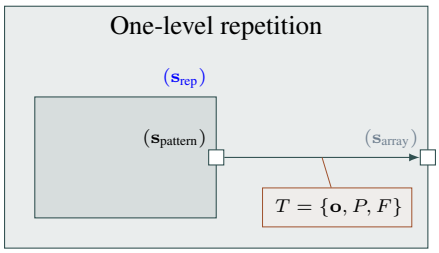


Figure 4. A one-level hierarchy

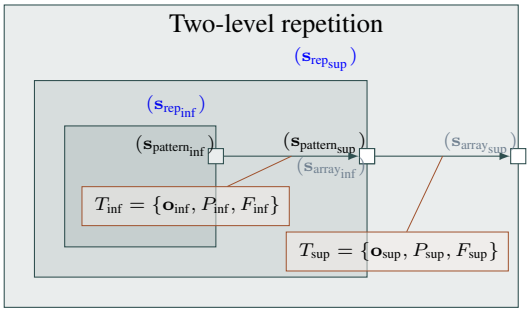


Figure 5. A two-level hierarchy

T_{inf}, and a common array ($\mathbf{s}_{\text{pattern}_{\text{sup}}} = \mathbf{s}_{\text{array}_{\text{inf}}}$).

$$\begin{aligned} &\forall \mathbf{r}_{\text{sup}}, \mathbf{0} \leq \mathbf{r}_{\text{sup}} < \mathbf{s}_{\text{rep}_{\text{sup}}}, \\ &\mathbf{ref}_{\mathbf{r}_{\text{sup}}} = \mathbf{o}_{\text{sup}} + P_{\text{sup}} \cdot \mathbf{r}_{\text{sup}} \bmod \mathbf{s}_{\text{array}_{\text{sup}}} \end{aligned} \tag{6}$$

$$\begin{aligned} &\forall \mathbf{r}_{\text{inf}}, \mathbf{0} \leq \mathbf{r}_{\text{inf}} < \mathbf{s}_{\text{rep}_{\text{inf}}}, \\ &\mathbf{ref}_{\mathbf{r}_{\text{inf}}} = \mathbf{o}_{\text{inf}} + P_{\text{inf}} \cdot \mathbf{r}_{\text{inf}} \bmod \mathbf{s}_{\text{array}_{\text{inf}}} \end{aligned} \tag{7}$$

Having the two tilers connected through a common array and using an Array-OL construction named "short-circuit"[2] that allows expressing direct relations between array elements through several connected tilers and considering the two repetitions like a single repetition, the relation becomes:

$$\begin{aligned} &\forall \begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix}, \mathbf{0} \leq \begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix} < \begin{bmatrix} \mathbf{s}_{\text{rep}_{\text{sup}}} \\ \mathbf{s}_{\text{rep}_{\text{inf}}} \end{bmatrix}, \\ &\mathbf{ref}_{\mathbf{r}_{\text{sup}}\mathbf{r}_{\text{inf}}} = \mathbf{o}_{\text{sup}} + F_{\text{sup}} \cdot \mathbf{o}_{\text{inf}} + (P_{\text{sup}}\ F_{\text{sup}} \cdot \\ &\qquad P_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix} \bmod \mathbf{s}_{\text{array}_{\text{sup}}} \end{aligned} \tag{8}$$

In both cases the relation can be writen under the form of equation 5.

Uniform dependences between repetitions. Taking equation 5 with an uniform dependence $\mathbf{d}$, according to the definition of an inter-repetition dependence (equation 2):

$$\begin{aligned} &\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{rep}}, \mathbf{r}_{\text{dep}} = \mathbf{r} - \mathbf{d} \\ &\mathbf{ref}_{\mathbf{r}_{\text{dep}}} = \mathbf{o} + P \cdot \mathbf{r}_{\text{dep}} \bmod \mathbf{s}_{\text{array}} \\ \Rightarrow &\mathbf{ref}_{\mathbf{r}} - \mathbf{ref}_{\mathbf{r}_{\text{dep}}} = P \cdot (\mathbf{r} - \mathbf{r}_{\text{dep}}) \\ \Rightarrow &\mathbf{d}_{\mathbf{ref}_{\mathbf{r}}} = \mathbf{ref}_{\mathbf{r}} - \mathbf{ref}_{\mathbf{r}_{\text{dep}}} = P \cdot \mathbf{d} \end{aligned} \tag{9}$$

Accordingly to equation 9, an uniform dependence between repetitions is equivalent to an uniform dependence between the references of these repetitions inside an array ($\mathbf{d}_{\mathbf{ref}_{\mathbf{r}}}$).

Analysis. We have shown that in both cases (one or two levels of hierarchy) we can express the relation between the repetitions and an array with a relation as shown in equation 5. Also equation 9 proves that an uniform dependence between repetitions is equivalent to a dependence between the references of these repetitions inside an array.

Furthermore, the constraint that says that the semantics of an application must remain the same after applying a transformation implies that the arrays at the border of the transformation's action must be produced in the same way, so all the references inside the array before the transformation must be present after the transformation. *An eventual uniform dependence between the references inside an array must also remain unchanged.* This is the link that we were looking for to connect the dependences between the repetitions before and after the transformation.

Now, having an initial structure expressed by the relation to an exterior array and an initial dependence between the references introduced by the dependence between the repetitions:

$$\begin{aligned} &\forall \mathbf{r}_{\text{before}}, \mathbf{0} \leq \mathbf{r}_{\text{before}} < \mathbf{s}_{\text{rep}_{\text{before}}}, \mathbf{ref}_{\mathbf{r}_{\text{before}}} = \\ &\qquad \mathbf{o}_{\text{before}} + P_{\text{before}} \cdot \mathbf{r}_{\text{before}} \bmod \mathbf{s}_{\text{array}} \end{aligned} \tag{10}$$

$$\mathbf{d}_{\mathbf{ref}_{\text{before}}} = P_{\text{before}} \cdot \mathbf{d}_{\text{before}} \tag{11}$$

and a final structure expressed by a similar relation:

$$\begin{aligned} &\forall \mathbf{r}_{\text{after}}, \mathbf{0} \leq \mathbf{r}_{\text{after}} < \mathbf{s}_{\text{rep}_{\text{after}}}, \mathbf{ref}_{\mathbf{r}_{\text{after}}} = \\ &\qquad \mathbf{o}_{\text{after}} + P_{\text{after}} \cdot \mathbf{r}_{\text{after}} \bmod \mathbf{s}_{\text{array}} \end{aligned} \tag{12}$$

$$\mathbf{d}_{\mathbf{ref}_{\text{after}}} = P_{\text{after}} \cdot \mathbf{d}_{\text{after}} \tag{13}$$

, by constraining the dependence between the references to remain the same, we have:

$$P_{\text{before}} \cdot \mathbf{d}_{\text{before}} = P_{\text{after}} \cdot \mathbf{d}_{\text{after}} \tag{14}$$

[2] Philippe Dumont's PhD [4], page 61

Solving equation 14 is enought to find the dependence(s) on the new repetitions. If there is no solution it means that there is no uniform dependence that can express on the new repetitions the same exact dependence as before the transformation, therefore the semantics of the application cannot be kept unchanged and therefore the transformation is not correct. If there is more than one solution for the equation, each solution corresponds to a dependence on the new repetition space.

If after the transformation we have just one level of hierarchy, each solution will be translated into a dependence on the repetition space. If we have two levels of hierarchy, each solution will be used to compute the dependences on the repetition spaces of each hierarchy level [3]:

$$\mathbf{d}_{\text{after}} = \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ \mathbf{d}_{\text{inf}} \end{pmatrix} \tag{15}$$

If the dependence on the superior level $\mathbf{d}_{\text{sup}}$ is null, for this solution we have no dependence on the superior level, and the dependence on the inferior level will be represented by the corresponding $\mathbf{d}_{\text{inf}}$. If $\mathbf{d}_{\text{sup}}$ is not null, we have dependences between elements on different blocks, represented by a dependence on the superior level.

The Array-OL semantics for inter-repetition dependences forces the passing of all the blocks containing depending elements to the inferior hierarchy level. The exact element-to-element dependence on the inferior level will be represented by a default link tagged with a tiler. The tiler will be represented by the exact corresponding output tiler of the inferior level, with a shifted origin.

Computing the shifted origin. The element-to-element dependence for repetitions in different blocks will be expressed as sum of the two dependences from the two levels of hierarchy. Having the depending repetitions in different blocks we don't need to express the inferior dependence with the use of the inter-repetition concept. Using a copy of the output tiler with a shifted origin in enough:

$$\forall\, \mathbf{r}_{\text{inf}}, \mathbf{0} \leq \mathbf{r}_{\text{inf}} < \mathbf{s}_{\text{rep}_{\text{inf}}}, \mathbf{ref}_{\mathbf{r}_{\text{def}}} = \mathbf{o}_{\text{def}} + P_{\text{inf}} \cdot \mathbf{r}_{\text{inf}} \tag{16}$$

The formula for computing the shifted origin can be obtained by imposing the constrain that the dependences remain the same as before the transformation, even when repetitions are in different blocks. For a repetition $\begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix}$ that depends on a repetition outside its block, we have the reference inside the original array:

$$\mathbf{ref}_{\mathbf{r}_{\text{sup}}\mathbf{r}_{\text{inf}}} = \mathbf{o}_{\text{sup}} + F_{\text{sup}} \cdot \mathbf{o}_{\text{inf}} + (P_{\text{sup}}\ F_{\text{sup}} \cdot P_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{r}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix} \bmod \mathbf{s}_{\text{array}_{\text{sup}}} \tag{17}$$

and the depending repetition:

$$\mathbf{ref}_{\mathbf{r}_{\text{dep}}} = \mathbf{o}_{\text{sup}} + F_{\text{sup}} \cdot \mathbf{o}_{\text{def}} + (P_{\text{sup}}\ F_{\text{sup}} \cdot P_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{r}_{\text{sup}} - \mathbf{d}_{\text{sup}} \\ \mathbf{r}_{\text{inf}} \end{pmatrix} \bmod \mathbf{s}_{\text{array}_{\text{sup}}} \tag{18}$$

thus giving the distance between the two:

$$\mathbf{d}_{\mathbf{ref}_{\mathbf{r}}} = \mathbf{ref}_{\mathbf{r}_{\text{sup}}\mathbf{r}_{\text{inf}}} - \mathbf{ref}_{\mathbf{r}_{\text{dep}}} = F_{\text{sup}} \cdot (\mathbf{o}_{\text{inf}} - \mathbf{o}_{\text{def}}) + (P_{\text{sup}}\ F_{\text{sup}} \cdot P_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ 0 \end{pmatrix} \tag{19}$$

Using the equation 13 in the case of a two-level hierarchy we get:

$$\mathbf{d}_{\mathbf{ref}_{\text{after}}} = (P_{\text{sup}}\ F_{\text{sup}} \cdot P_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ \mathbf{d}_{\text{inf}} \end{pmatrix} \tag{20}$$

and therefore

$$\begin{aligned} &\mathbf{d}_{\mathbf{ref}_{\mathbf{r}}} = \mathbf{d}_{\mathbf{ref}_{\text{after}}} \\ \Rightarrow &(P_{\text{sup}}\ F_{\text{sup}} \cdot P_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ \mathbf{d}_{\text{inf}} \end{pmatrix} = F_{\text{sup}} \cdot (\mathbf{o}_{\text{inf}} - \mathbf{o}_{\text{def}}) + (P_{\text{sup}}\ F_{\text{sup}} \cdot P_{\text{inf}}) \cdot \begin{pmatrix} \mathbf{d}_{\text{sup}} \\ 0 \end{pmatrix} \\ \Rightarrow &P_{\text{sup}} \cdot \mathbf{d}_{\text{sup}} + F_{\text{sup}} \cdot P_{\text{inf}} \cdot \mathbf{d}_{\text{inf}} \\ &\quad = F_{\text{sup}} \cdot (\mathbf{o}_{\text{inf}} - \mathbf{o}_{\text{def}}) + P_{\text{sup}} \cdot \mathbf{d}_{\text{sup}} \\ \Rightarrow &P_{\text{inf}} \cdot \mathbf{d}_{\text{inf}} = \mathbf{o}_{\text{inf}} - \mathbf{o}_{\text{def}} \end{aligned} \tag{21}$$

As result, the shift of the origin will be computed using $\mathbf{d}_{\text{inf}}$:

$$\mathbf{o}_{\text{def}} = \mathbf{o}_{\text{inf}} - P_{\text{inf}} \cdot \mathbf{d}_{\text{inf}} \tag{22}$$

where $\mathbf{o}_{\text{def}}$ represents the origin of the tiler of the default link, $\mathbf{o}_{\text{inf}}$ and P_{inf} the origin vector and paving matrix of the corresponding output tiler of the inferior level of hierarchy.

6 Conclusion

We aimed in this paper to analyze the interaction between the high-level transformations designed around Array-OL model of specification and the inter-repetition dependence extension. The extension was introduced in order to allow the construction of self-loops in the task-graph, the only way

[3] The separation between the two dependences is done accordingly to the size of the repetition spaces.

to define dependence relations between elements of a same array and to keep state information. The Array-OL specification model together with this extension is 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 and hierarchy.

As a transformation has an impact on maximum two successive hierarchy levels of repetitions, we have shown how, having the structure of the application before and after the transformation, we can compute the new dependences that will express the same element-to-element dependences as before the transformation. This guarantees that the semantics of an application remain unchanged. Based on this specification, an algorithm for adapting dependences was proposed and proved. The algorithm works independently from the Array-OL transformations, which facilitated the implementation.

The concepts of Array-OL are at the core of our model-driven engineering framework Gaspard2 [3] designed to codesign intensive signal processing applications on system-on-chip. The specification language of Gaspard2 can be seen as a subset of MARTE, efforts being made to make the two fully compatible [11].

The Array-OL transformations (without inter-repetition dependences) were already formalized and implemented in our tools. Following this study and in order to validate the results, an extension of the transformation tool was implemented, in order to take into account inter-repetition dependences. The results confirmed the validity of our algorithm and they were integrated in Gaspard2.

References

[1] A. Amar, P. Boulet, and P. Dumont. Projection of the Array-OL specification language onto the Kahn process network computation model. In *International Symposium on Parallel Architectures, Algorithms, and Networks*, Las Vegas, Nevada, USA, Dec. 2005.

[2] P. Boulet. Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing. Research Report RR-6467, INRIA, Mar. 2008.

[3] DaRT Team. Graphical Array Specification for Parallel and Distributed Computing (GASPARD2). `http://www.gaspard2.org/`, 2009.

[4] P. Dumont. *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, Dec. 2005.

[5] P. Dumont and P. Boulet. Another multidimensional synchronous dataflow: Simulating Array-OL in ptolemy II. Research Report RR-5516, INRIA, Mar. 2005.

[6] C. Glitia and P. Boulet. High level loop transformations for multidimensional signal processing embedded applications. In *SAMOS 2008 Workshop*, Samos, Greece, July 2008.

[7] C. Glitia, P. Dumont, and P. Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 2009.

[8] J. Keinert, C. Haubelt, and J. Teich. Modeling and analysis of windowed synchronous algorithms. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages III–892– III–895, 2006.

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

[10] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, Aug. 2002.

[11] E. Piel, R. B. Attitalah, P. Marquet, S. Meftali, S. Niar, A. Etien, J.-L. Dekeyser, and P. Boulet. Gaspard2: from MARTE to SystemC simulation. In *Modeling and Analyzis of Real-Time and Embedded Systems with the MARTE UML profile DATE'08 Workshop*, Mar. 2008.

[12] 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).