Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 561863, 22 pages
doi:10.1155/2008/561863

Research Article

Synchronous Modeling and Analysis of Data Intensive Applications

Abdoulaye Gamatié,[1] Éric Rutten,[2] Huafeng Yu,[1] Pierre Boulet,[1] and Jean-Luc Dekeyser[1]

[1] *LIFL, CNRS/INRIA, Université de Lille 1, Parc de la Haute Borne, Bât A 40 avenue Halley, 59650 Villeneuve d'Ascq Cedex, France*
[2] *INRIA Rhône-Alpes, 655 avenue de l'Europe, Montbonnot, 38334 Saint-Ismier cedex, France*

Correspondence should be addressed to Abdoulaye Gamatié, abdoulaye.gamatie@lifl.fr

Received 4 July 2007; Revised 8 March 2008; Accepted 25 June 2008

Recommended by Marc Pouzet

We present the modeling of data-intensive parallel applications following the synchronous approach. We consider the GASPARD environment, which is dedicated to high-performance system-on-chip (SoC) codesign. Our motivation is to bridge the gap between the GASPARD design approach and the formal validation techniques provided by the synchronous technology. First, we define a synchronous dataflow equational model of GASPARD models. The modeling formalism adopted in GASPARD consists of an extension of the domain-specific language Array-OL. Then, we address correctness issues (e.g., causality and synchronizability analyses) about GASPARD models via their corresponding synchronous descriptions in order to formally validate the original system descriptions.

1. INTRODUCTION

Computing and analyzing large amounts of data play an increasingly important role in embedded systems. The concerned applications often perform computations on regular multidimensional data structures. Typical examples are state-of-the-art multimedia applications that require high-performance (e.g., high-definition TV, medical imaging, radar or sonar signal processing, telecommunications, mobile cell phones, etc.). The highly desirable design approaches for such applications are those providing users with well-adapted concepts in order to represent the data manipulations, and the techniques that trustworthily guarantee important implementation requirements.

The domain-specific language Array-OL has been originally proposed within an industrial context by Thomson Marconi Sonar, now Thales, for the description of data-intensive applications manipulating multidimensional data structures [1]. It offers adequate concepts to describe both task parallelism and data-parallelism in applications. Furthermore, Array-OL descriptions are platform-independent. All these features make Array-OL very expressive and suitable for the specification of data-intensive applications. This is the reason why they have been adopted in *graphical array specification for parallel and distributed computing* (GASPARD) [2], an integrated development environment dedicated to the modeling, simulation, testing, verification, and code generation of high-performance system-on-chip (SoC).

On the other hand, the *synchronous approach* [3] has been defined to provide embedded real-time system designers with formal concepts that favor the trusted design. Its basic assumption is that computation and communication are instantaneous, referred to as the "synchrony hypothesis." The execution of a system is seen through the chronology and simultaneity of observed events. This is a main difference from visions where the system execution is rather considered under its chronometric aspect, that is, duration has a significant role. This assumption confers to synchronous languages a deterministic semantics in presence of concurrency.

The combination of the advantages of both Array-OL and synchronous languages within a unique design framework can significantly increase the ability of developers of high-performance applications to (i) adequately describe such applications, and (ii) trustworthily guarantee the correctness of the corresponding implementations. The aim of this paper is mainly to answer this demand by bridging the gap between Array-OL-based techniques and the synchronous approach.

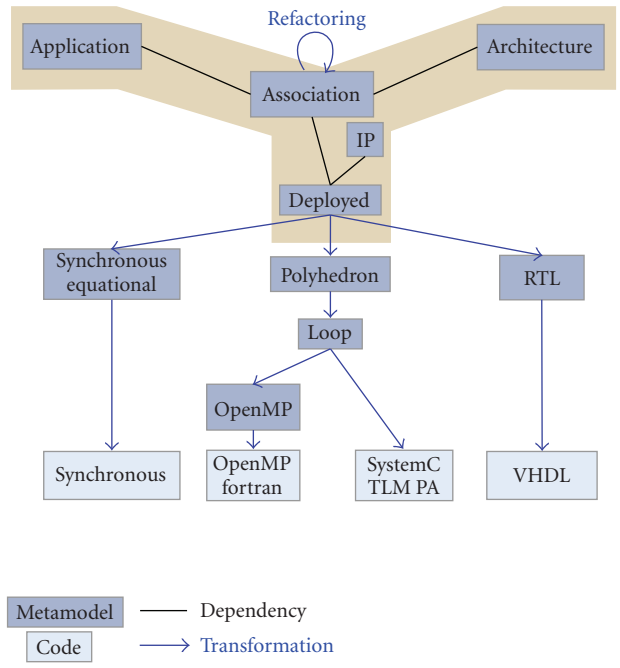


Figure 1: The GASPARD design methodology.

1.1. Rationale: the GASPARD framework

Our proposition is defined within the GASPARD framework [2] that is dedicated to the development of high-performance SoC and intensive signal processing applications. The modeling formalism of GASPARD, referred to as GASPARD *profile*, consists of a UML-style extension of the Array-OL language with additional concepts that are necessary to represent different aspects of the systems to be designed: software (or application) part, hardware (or architecture) part, association of the two, and deployment of the resulting associated model on specific platforms using a library of *intellectual properties* (IP).

GASPARD promotes a software/hardware codesign methodology based on *model-driven engineering* (MDE) as illustrated by Figure 1: the software and hardware parts of the system are first modeled, then refined toward lower level languages for various purposes: formal validation with synchronous languages [3] (currently Lustre and Signal), simulation in SystemC and OpenMP Fortran, and circuit synthesis in VHDL. At each level of this refinement, the concepts are characterized by a dedicated metamodel, and the transitions from one level to another are obtained via automatic model transformations. The backbone environment that implements this methodology is Eclipse.

Through the methodology adopted in the GASPARD framework, various design issues can be addressed at different abstraction levels depending on the suitability of these levels to enable the required analysis. In addition, one can notice that during the transformation of high-level GASPARD models toward lower levels, for example, SystemC, OpenMP Fortran or synchronous languages, there could be a loss of abstraction. But it does not matter since the low-level descriptions resulting from the transformation constitute approximations that preserve the properties of interest in the initial GASPARD models. In particular, these descriptions are expressive enough to be considered for functional and nonfunctional simulation in SystemC and OpenMP Fortran, circuit synthesis in VHDL and formal verification with synchronous languages.

From a practical point of view, the use of MDE approach to implement the GASPARD methodology strongly favors *reusability and separation of concerns*, which represent real benefits when designing complex systems. Indeed, MDE allows us to access useful pre-existing facilities (simulation, performance analysis, formal verification, etc.) depending on the target representations reached via the model transformations.

Most of the design concepts of GASPARD have been integrated in the OMG standard MARTE profile (http://www.omgmarte.org/), dedicated to the *modeling and analysis of real-time and embedded systems*. These concepts are defined within the package named *repetitive structure modeling* (RSM), in the MARTE specification.

In this paper, we consider the specification and the modeling of high-performance applications. These applications are initially designed with the GASPARD extension of Array-OL. The resulting descriptions are translated into a set of synchronous dataflow equations to analyze various aspects of the applications. Then, we address some correctness issues on the obtained descriptions with the help of the synchronous technology: causality analysis, single assignment, synchronizability resulting from the deployment of a platform, and so forth.

In the sequel, we give an introduction to Array-OL and the synchronous language Signal, which is used here for illustration, followed by a presentation of some related work. Section 2 describes how GASPARD models are translated into a set of synchronous dataflow equations, based on two possible interpretations of data parallelism in target applications. The usefulness of such a translation is shown in Section 3, where we use some specific concepts of the synchronous approach, such as affine clocks, in order to check some important properties in GASPARD models of considered applications. Finally, Section 4 gives the concluding remarks and future work.

1.2. Array-OL: the underlying specification language of GASPARD

Array-OL [1, 4, 5] is a mixed graphical-textual language that enables to specify both *task parallelism* and *data parallelism* available in intensive signal processing applications. Among the basic characteristics of this domain-specific language, we mention the following ones.

(i) *Data dependency expressions*: Array-OL only specifies true data dependencies in order to express the full parallelism of the application. In such a way, except the minimal partial order resulting from the specified data dependencies, no other order is a priori assumed.

(ii) *Functionally-deterministic specifications*: any execution schedule that respects the data dependencies

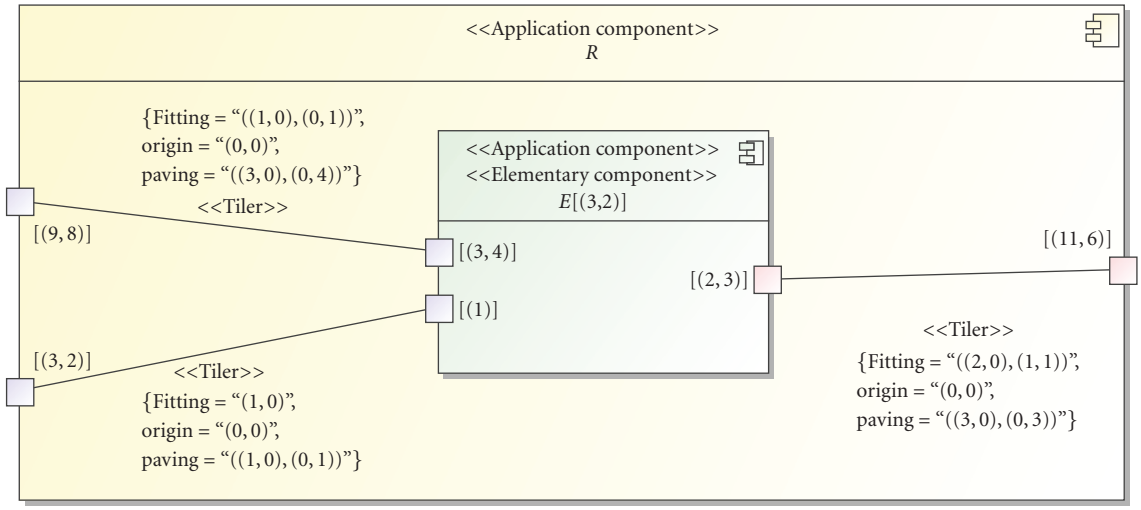


Figure 2: An Array-OL repetitive task depicted in the GASPARD framework.

specified in Array-OL necessarily leads to the same results.

(iii) *Single assignment*: the language handles values, not variables, so a value is produced only once.

(iv) All values are manipulated under the form of *multi-dimensional arrays*, with a possible *infinite dimension* representing time. These arrays are also *toroidal*. Indeed, some spatial dimensions may represent some physical tori (e.g., hydrophones around a submarine), and some frequency domains obtained by fast fourier transformations (FFTs) are toroidal. The language is not a dataflow language as the dimensions of the arrays can be mapped on space (several processors) or time for an execution. No hypothesis on this mapping is done at specification time.

As an informal overview of the basic concepts of Array-OL, let us consider the visual description given in Figure 2. It represents a task R that takes as inputs a $(9,8)$-array and a $(3,2)$-array, and produces an $(11,6)$-array. R expresses a data-parallel repetition in the sense that the same function E is applied to subsets of the input data in the form of $(3,4)$-array and a (1)-array to produce subsets of the output data in the form of $(2,3)$-array. The way each subset of data is either extracted or stored in the input and output arrays is determined by using the information attached to the links stereotyped "*Tiler*," which connect two different ports. Array-OL defines three kinds of tasks: elementary tasks (e.g., E when it is an elementary function), repetitive tasks (e.g., R) that express data-parallelism, and hierarchical tasks, which hierarchically combine tasks through a task graph. The remainder of this section mainly presents each kind of these tasks and clarifies the unexplained details of Figure 2.

Elementary tasks

Such a task has a body corresponding to an atomic computation block that typically consists of a function (e.g., addition, dot product, FFT). The interface of this function must comply with the interface of the task characterized by ports. In Figure 2, E may be considered as an elementary task. The shape of its input arrays (*resp.*, output arrays) is noted on the corresponding ports: $(3,4)$ and (1) (*resp.*, $(2,3)$).

Repetitive tasks

Before presenting this task, we must first precise what the term "repetition" stands for. A repetition enables to execute a set of operations in an arbitrary order, including parallel, without any computational differences in the result (no semantic change). A repetitive task (e.g., task R in Figure 2) expresses the data parallelism in Array-OL by specifying a repetition on the elements of the input and output arrays. Each operation is achieved by a task *instance*, which executes independently of the other instances. The subarrays consumed and produced by repeated task instances (e.g., instances of task E in Figure 2) have the same shape. They are referred to as *patterns* when considered as inputs/outputs of a task instance, and *tiles* when considered as a set of elements within incoming/outgoing arrays of the repetitive task. Such tiles are regularly spaced sets of regularly stored elements, hence their representation as subarrays. Note that by abuse of language, the terms "pattern" and "tile" are often used to mean the same thing.

The way the tiles is constructed is defined via *tilers*, which are associated with each array (i.e., each edge in the graphical representation). A tiler extracts (resp., stores) tiles from (resp., in) an array based on some information: F a *fitting* matrix (how array elements fill the tiles), $\mathbf{o}$ the *origin* of the *reference tile* (for the *reference repetition*), and P a *paving* matrix (how the tiles cover arrays).

The *repetition space* indicating the number of task instances is itself defined as a 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 nested parallel loops. In Figure 2, the shape of repetition space is $(3, 2)$.

Given a tile, let its *reference element* denote its origin point from which all its other elements can be extracted. The *fitting* matrix is used to determine these elements. Their coordinates, represented by $\mathbf{e_i}$, are built as the sum of the coordinates of the reference element and a linear combination of the fitting vectors, modulo the size of the array (since arrays are toroidal) as follows:

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \quad \mathbf{e_i} = \mathbf{ref} + F \times \mathbf{i} \bmod \mathbf{s}_{\text{array}}, \tag{1}$$

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

Figure 3 shows an example of fitting corresponding to the single output array in task R of Figure 2. Here, there are 6 elements in this tile since the shape of the pattern is $\binom{2}{3}$. The reference element is represented by vector $\binom{0}{0}$. The indexes of the remaining elements are thus $\binom{0}{1}$, $\binom{0}{2}$, $\binom{1}{0}$, $\binom{1}{1}$, and $\binom{1}{2}$. The positions of these elements in the tile relative to the reference point are determined as follows:

$$\begin{aligned}
F \times \begin{pmatrix} 0 \\ 0 \end{pmatrix} &= \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \\
F \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} &= \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \\
F \times \begin{pmatrix} 0 \\ 2 \end{pmatrix} &= \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \\
F \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} &= \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \\
F \times \begin{pmatrix} 1 \\ 1 \end{pmatrix} &= \begin{pmatrix} 3 \\ 1 \end{pmatrix}, \\
F \times \begin{pmatrix} 1 \\ 2 \end{pmatrix} &= \begin{pmatrix} 4 \\ 2 \end{pmatrix}.
\end{aligned} \tag{2}$$

For each repetition, one needs to specify the reference elements of the input and output tiles.

A similar scheme as the one used to enumerate the elements of a tile 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:

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \quad \mathbf{ref_r} = \mathbf{o} + P \times \mathbf{r} \bmod \mathbf{s}_{\text{array}}, \tag{3}$$

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. Figure 4 illustrates an example of paving corresponding to the first input of task R, that is, the (9×8)-array.

Hierarchical tasks

They are represented by hierarchical acyclic graphs in which each node consists of an Array-OL task, and edges are labelled by the arrays exchanged between these nodes. This naturally leads to hierarchical description of tasks (see Figure 9 for illustration).

An execution model for Array-OL

GASPARD considers a particular execution model according to which Array-OL-based descriptions are executed. This model of execution defines an executable Array-OL description as a hierarchical task model in which the top-level is composed of a single task that plays a similar role as the "main" in a C program. A transformation of Array-OL specifications, called *fusion* [6], allows one to automatically transform any task model into such a hierarchical task model. Infinite arrays are only manipulated at the top-level of the hierarchy (see Figure 9 for illustration). Furthermore, the infinite dimension is interpreted as a temporal dimension. As a result, the unique top-level task in the hierarchy receives its input arrays through a *flow* and produces its output arrays following the arrival order of its inputs. In the sublevels of the hierarchy, the arrays received at each step of the flow are manipulated as usual in Array-OL, that is, without any consideration of temporal dimension. Every subtask is assumed to compute its outputs only after all the inputs have been received. In other words, every output of a sublevel task depends on all its input. In GASPARD, an *application* is represented by such a hierarchical task. In this paper, all applications are considered with respect to this execution model.

Behavior of GASPARD programs

We are interested in the behavior of GASPARD programs in terms of the sets of *computations* they define, that is, *what functions are applied to what data*.

Hence, the behavior of a GASPARD program can be defined as follows:

(i) for an elementary task E: the singleton of one computation, applying the function f corresponding on the input data i to produce the output data o, hence behavior$(E) = \{o = f(i)\}$;

(ii) for a repetitive task R: the union of the sets defining the behavior of each of the $|\mathbf{r}|$ instances of the computations of the repeated body, each applying to the appropriate patterns of data in input and in output, as determined by the tilers, hence behavior$(R) = \cup_{k \in 1 \cdots |\mathbf{r}|} B_k$, where B_k denotes the set of computations for each instance; for example, if the repeated body is an elementary task applying function f, and the input array i is decomposed by the input tiler into patterns p_i^k, and the output array o is recomposed by the input tiler from patterns p_o^k, then $\cup_{k \in 1 \cdots |\mathbf{r}|} \{p_o^k = f(p_i^k)\}$;

(iii) for a hierarchical task H: the union of the sets defining the behavior of the subtasks in the body, each applying to the appropriate data in input and in output, as determined by the dependencies, hence behavior$(H) = \cup_{t \in \text{Tasks}} B_t$; for example, a hierarchical task T with input i and output o, having two elementary subtasks T_1 with input i and output l, and T_2 with input l and output o, has the behavior: $\{o = f_2(l), l = f_1(i)\}$.

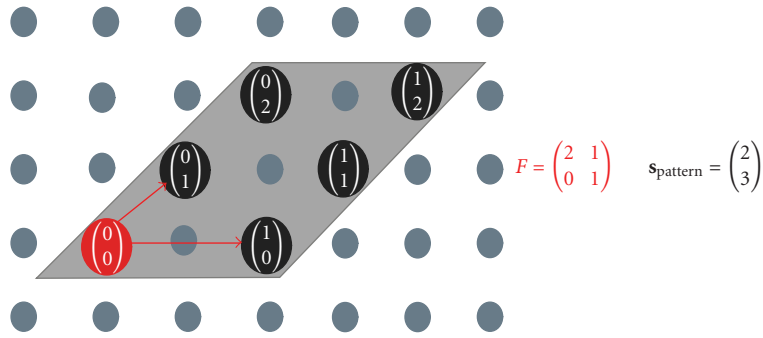


Figure 3: A fitting example.

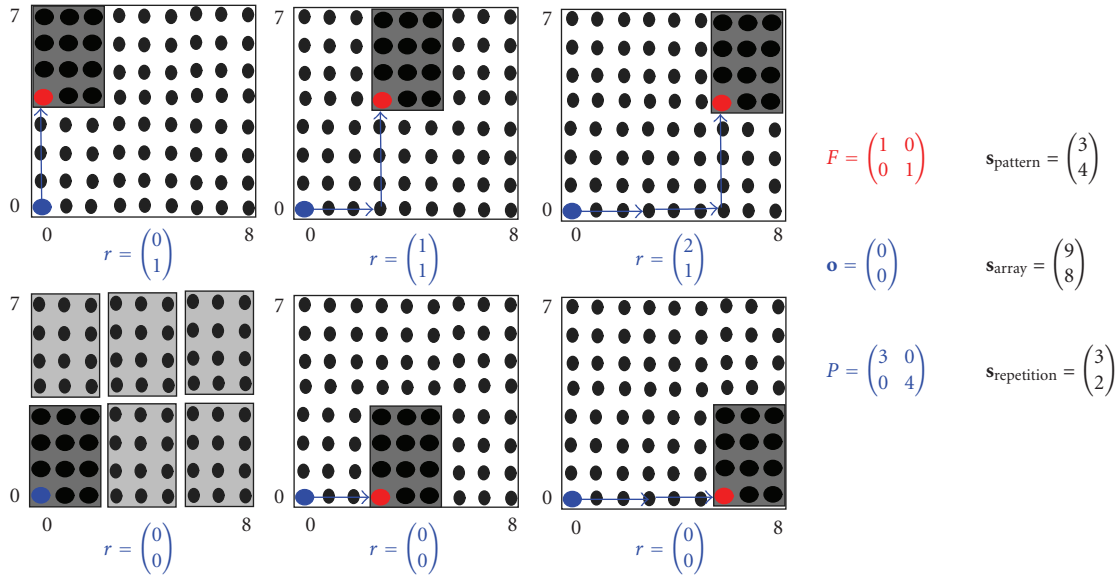


Figure 4: Paving example: a 2D pattern tiling perfectly a 2D array.

1.3. The synchronous language Signal

The synchronous language signal [7] belongs to the family of dataflow languages whose origin can be historically associated with earlier studies on dataflow models started in 70s [8–10]. It is the case of the synchronous dataflow languages Lustre [11] and Lucid Synchrone [12]. Signal handles unbounded series of typed values $(x_\tau)_{\tau\in\mathbb{N}}$, called *signals*, denoted as x and implicitly indexed by discrete time. At a given instant, a signal may be present, at which point it holds a value; or absent and denoted $\perp$. The set of instants where a signal x is present represents its *clock*, noted $\hat{}x$. Two signals x and y, which have the same clock are said to be *synchronous*. A *process* (or a *node*) is a system of equations over signals that specifies relations between values and clocks of the signals. A *program* is a process. Signal relies on the following six primitive constructs:

Relations

$y := f(x1,\ldots,xn) \stackrel{\text{def}}{\equiv}$ for all $\tau \geq 0 : y_\tau \neq \perp \Leftrightarrow x1_\tau \neq \perp \Leftrightarrow \cdots \Leftrightarrow xn_\tau \neq \perp$, and $y_\tau = f(x1_\tau,\ldots,xn_\tau)$.

Delay

$y := x \$ 1 \; init \; c \stackrel{\text{def}}{\equiv}$ for all $\tau \; x_\tau \neq \perp \Leftrightarrow y_\tau \neq \perp$ and for all $\tau > 0 : y_\tau = x_k$, where $k = \max\{\tau' \mid \tau' < \tau \text{ and } x_{\tau'} \neq \perp\}$, $y_0 = c$.

Undersampling

$y := x \; when \; b$ where b is Boolean $\stackrel{\text{def}}{\equiv}$ for all $\tau \geq 0 : y_\tau = x_\tau$ if $b_\tau =$ true, else $y_\tau = \perp$. The expression $y := when \; b$ is equivalent to $y := b \; when \; b$.

Deterministic merging

$z := x \; default \; y \stackrel{\text{def}}{\equiv}$ for all $\tau \geq 0 : z_\tau = x_\tau$ if $x_\tau \neq \perp$, else $z_\tau = y_\tau$.

Composition

$(|P|Q|) \stackrel{\text{def}}{\equiv}$ union of the equations of P and Q. This operator is commutative and associative.

```
(1) process k_Overspl = {integer k;}
(2)  (? event c1; ! event c2; )
(3)  (| cnt := (k − 1 when c1) default (pre_cnt − 1)
(4)   | pre_cnt := cnt $ 1 init 0
(5)   | c1^= when (pre_cnt <= 0)
(6)   | c2 := when (^cnt)
(7)   |)
(8)   where integer cnt, pre_cnt;
(9) end; %process k_Overspl%
```

```
c1 : tt ⊥ ⊥ ⊥ tt ⊥ ⊥ ···
cnt : 3 2 1 0 3 2 1 ···
pre_cnt : 0 3 2 1 0 3 2 ···
c2 : tt tt tt tt tt tt tt ···
```

Figure 5: A clock oversampling process in signal and a trace.

Table 1: Comparison of various specification languages for signal processing.

Language	Data form	Data access style	Access generality		Control structures			Tool support
			sliding window	under/over sampling	delays	hierarchy	modes	
SDF [24]	1-D	subarray	n	n	y	n	n	Ptolemy
CSDF [25]	1-D	subarray	n	n	y	n	y	y
Stream-It [20]	1-D	subarray	y	y	y	y	n	StreamIT
MDSDF [26]	$*$-D	subarray	n	y	y	n	n	Ptolemy
GMDSDF [27]	$*$-D	subarray	n	y	y	n	n	n
WSDF [28]	$*$-D	subarray	y	y	y	n	n	u
BLDF [29]	$*$-D	subarray	n	y	y	y	y	y
Array-OL [1, 4]	cyclic $*$-D	subarray	y	y	e	y	e	GASPARD
Alpha [23]	P	affine functions	y	y	y	y	n	MM-Alpha

• In column Data form, "1-D" means that the scheduling considers monodimensional data streams (that may carry multidimensional arrays); "$*$-D" means that these data streams are replaced by multidimensional arrays; "cyclic $*$-D" means that some dimensions of these multidimensional arrays may be cyclic; and "P" means that the language handles convex polyhedra of integer points.
• The following convention holds: "y:" supported feature, "n:" not supported feature, "e:" feature supported through extended constructs; "u:" unknown to us.
• In column Tool support, when the name of a tool is known, it is explicitly mentioned.

Hiding

P where $x \stackrel{\text{def}}{\equiv} x$ is local to the process P.

These constructs are expressive enough to derive other constructs for comfort and structuring. For instance, given two signals x and y, the extended construct $\hat{=}$ is used to specify that they are synchronous [13], noted $x\hat{} = y$. Signal offers a process frame that enables the definition of subprocesses (declared in the *where* subpart, see Figures 12 and, 13, e.g.). Subprocesses that are only specified by an interface without internal behavior are considered as external, and may be separately compiled processes or physical components. A useful notion of Signal is the *oversampling* mechanism. It consists of a temporal refinement of a given clock c_1, which yields another clock c_2, which is faster than c_1, meaning that c_2 contains more instants than c_1.

In the Signal process given in Figure 5, called *k_Overspl*, *k* is a constant integer parameter (line 1). The clock signals *c*1 and *c*2, respectively, denote input represented by "?" and output represented by "!" (line 2). Here, *c*2 is a 4-oversampling of *c*1. The *event* type is associated with clocks. It is equivalent to Boolean type where the only taken value is *true*. The local signals *cnt* and *pre_cnt* serve as counter to define 4 instants in *c*2 per instant in *c*1 lines (3 and 4).

There is a graphical syntax of Signal that is very similar to block diagrams. In such a syntax, a box represents a process and a connection between boxes represents the communication of signal values between processes (see illustrations in Figures 8 and 11).

1.4. Related work

Several languages have been proposed to deal at a high level of abstraction with multidimensional arrays, mostly for parallel scientific computing. The most well-known is high performance Fortran which uses an array notation to define compactly subarrays and that proposes parallel loops constructs and regular data distributions [14]. More recent efforts are hierarchically tiled arrays [15, 16], the cascade high-productivity language [17], Fortress [18] or the java-like X10 [19]. The tiling construct proposed by Array-OL is more adapted to signal processing applications, allowing overlapping tiles and modular accesses. Another difference is that Array-OL specifications manipulate time and space dimensions in the same way, allowing a specification that is independent of the execution strategy. Loop transformations are provided to adapt the specification of the algorithm to a particular hardware and execution strategy. The mapping and scheduling formalism proposed in GASPARD is strictly more expressive than the one of HPF and allows separating the mapping of the data and the mapping of the computations.

Table 1 gives a synthetic comparison of Array-OL with other popular specification languages for signal processing. Array-OL is the only one able to deal with the following

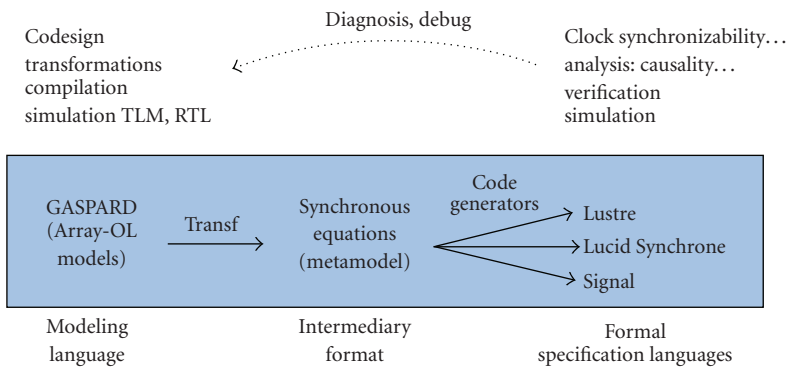


Figure 6: A global view of our approach.

requirements of the application domain:

(i) access to multidimensional arrays by regularly spaced subarrays;

(ii) ability to deal with sliding windows;

(iii) ability to deal with cyclic array dimensions;

(iv) ability to sub/oversample the arrays;

(v) hierarchical specification to deal with complex systems.

Stream processing languages such as StreamIt [20] and the synchronous dataflow (SDF) family of Ptolemy II [21] are usually not multidimensional with the notable exception of the multidimensional SDF of Ptolemy and its extensions (GMDSDF, WSDF, and BLDF). The Alpha language [22, 23] is an interesting alternative to Array-OL. However, this language uses loop index formulas to define the computations and not the higher level tiling construct of Array-OL that favors modularity.

Note that all these languages are able to statically manipulate unbounded sets of values. The unboundedness is often induced by the (infinite) temporal dimension in manipulated data structures. In the case of Alpha, the polyhedra can represent unbounded set of values. While each of them enables to describe dataflow specifications, the way the control aspects are described by varies with the language (see Table 1).

Finally, we note that synchronous languages also defines arrays in order to deal with specific algorithms and architectures. In [30], authors introduced arrays in Lustre in order to design and simulate systolic algorithms. This work leads to the implementation of their result on circuits [31]. More recently, an efficient compilation of array in Lustre programs has been proposed [32]. The Signal language also manipulates arrays in order to describe regular algorithms and architectures. In particular, the notion of array of processes [13] has been introduced in the language, which is well suited to model systolic algorithms. The notion of array is however less expressive in synchronous languages than in Array-OL. For instance, in synchronous languages, arrays are necessarily finite; as a result the time dimension cannot be considered as an array dimension unlike in Array-OL. In our translation of GASPARD models toward synchronous languages, an infinite array will be represented by a flow of finite subarrays.

2. FROM GASPARD TO SYNCHRONOUS LANGUAGES

2.1. Motivation elements for a translation toward synchronous languages

The goal of the translation presented in this section is to provide GASPARD designers with the possibility to address some correctness issues in the described models using the existing formal tools and techniques offered by the synchronous technology. Of course some of these issues may be addressed at the GASPARD formalism level with appropriate tools. Here, we rather propose a bridge between the GASPARD environment and the existing synchronous technology in order to achieve the analysis of model correctness. This is achieved according to the MDE philosophy adopted by GASPARD.

The first reason for this choice is that the development of required tools at the GASPARD formalism level can be very complex. For instance, checking the single assignment property of GASPARD model can be done by using sophisticated techniques such as [33, 34], which are achieved with polyhedra and linear programming as shown in [5]. However, such a solution may reveal penalizing due to the complexity induced by possible combinatorial explosion when manipulating polyhedra (even if tools like the *parametric integer programming*—PIP—solver (http://www.piplib.org/) are shown to be promising against the problems related to polyhedra manipulation). Provided a synchronous description corresponding to an abstraction (or an approximation) of a GASPARD model, checking single assignment on such an abstract description is straightforward and costless with compilers of synchronous languages.

The same observation holds when one wants to address further properties about GASPARD models, such as determinism and causality. The algorithms proposed in Feautrier's work on dataflow analysis of array and scalar [34] can be also considered as a possible solution to deal with such properties. However, the scalability of these algorithms remains a challenging issue.

Another important reason to take synchronous languages as the target representation is their capability to address the above properties in presence of complex control structures. The previously mentioned polyhedra-based techniques mainly deal with data dependencies. Adding control features to the GASPARD design model [35] leads to more difficult analysis of design properties. Synchronous languages offer a suitable model representation in which both data dependencies and control flow are analyzable uniformly. In the synchronous language Signal, the structure that serves for this purpose is the hierarchized conditional dependency graph (HCDG) [7]. Such a structure combines data dependencies and activation clocks that indicate when edges and vertexes of the dependency graph are valid with respect to the control flow. For instance, the mutual exclusion of different statements in a program specification is trivially detected on the HCDG. The analysis of causality, determinism, and single assignment also relies on this structure.

On the other hand, the synchronous technology provides us with typical techniques and tools that are proved to be very efficient to answer critical questions about the correct interaction of system components. For instance, in the methodology illustrated by Figure 1, after the deployment phase, the designer may need to check the synchronization between parts of the modeled system, deployed on different processors. Such a question could be answered with the clock synchronizability notions defined in synchronous languages [36, 37].

2.2. Global picture of the translation

Our translation approach is illustrated by Figure 6. We propose a transformation path toward synchronous technologies so as to be able to formally address new issues about the design correctness of GASPARD models, for example, causality and synchronizability properties. This transformation called *transf* is followed by a code generation phase that targets the synchronous dataflow languages Lustre, Lucid Synchrone, and Signal. Although our work covers the whole gray box in the figure, the real main contribution is the definition of *transf* and the analysis of the synchronous models based on this transformation. Synchronous dataflow equations and their translation to existing languages are in general well-understood topics.

Through the proposed approach, the advantages of both data-parallel and synchronous technologies are put together so as to benefit from their specific strengths: on the one hand, high-performance SoC design with GASPARD and on the other hand, formal analysis with synchronous tools. All this is well integrated within the GASPARD methodological framework [38].

In the next sections, we explain the basic modeling ideas on which the transformation *transf* relies. We first show a synchronous model that fully preserves the data parallelism and task parallelism of the GASPARD models (Section 2.3). Then, we present a refined version of such a model, where computations are serialized (Section 2.4). This second version can be typically associated with a monoprocessor execution platform. Finally, we briefly discuss the combination of both models within a mixed description that can feature a multiprocessor execution. Although our approach targets all the synchronous dataflow languages, we restrict ourselves to Signal for illustrations in this paper. The reader should therefore keep in mind that the resulting descriptions are the same for other languages. Moreover, we will use both graphical and textual (pseudo-Signal) notations to facilitate the understanding of the translation, and will fully describe some examples in the Signal syntax.

We concentrate on the modeling of computation, as it is the heart of our contribution, rather than modeling of data structures.

Task	::=	Interface; Body	(r1)
Interface	::=	i, o : {Port}	(r2)
Port	::=	id; type; shape	(r3)
Body	::=	$\text{Body}^h \mid \text{Body}^r \mid \text{Body}^e$	(r4)
Body^e	::=	some function f	(r5)
Body^r	::=	t_i, t_o : {Tiler}; (**r**; Task);	(r6)
Connexion	::=	p_i, p_o : Port	(r7)
Tiler	::=	Connexion; (F; **o**; P)	(r8)
Body^h	::=	{Task}; {Connexion}	(r9)

Figure 7: A grammar of GASPARD concepts.

2.3. Parallel model

The transformation from the GASPARD source model to the synchronous model is structural, following the syntactical constructs. It is greatly facilitated by the similarity between GASPARD and Signal since both have a recursive block-diagram structure.

We present the transformation by concentrating on aspects related to the behavior in terms of the sets of computations, and, for the sake of clarity, skip technical details on, for example, constructing interfaces or connecting nodes according to data dependencies.

2.3.1. Structural transformation

The grammar given in Figure 7 describes the basic GASPARD specification concepts. By convention, the notation $x : X$ in the grammar means that X is the type of x, and $\{X\}$ denotes a set of objects of type of X.

The recursive algorithm following the grammatical structure is as follows, starting with (r1).

(r1) Each GASPARD task is represented by a Signal process or node, with an interface according to (r2), and a body translating the GASPARD task body according to (r4).

(r2) Each input and output array of an interface is translated according to (r3).

(r3) Each port is translated as a Signal flow. As mentioned previously, in Array-OL, arrays can have an infinite dimension, and there is no explicit representation of time whatsoever. This infinite dimension of an array can be suitably represented by an infinite flow of arrays of the remaining dimensions as stated in

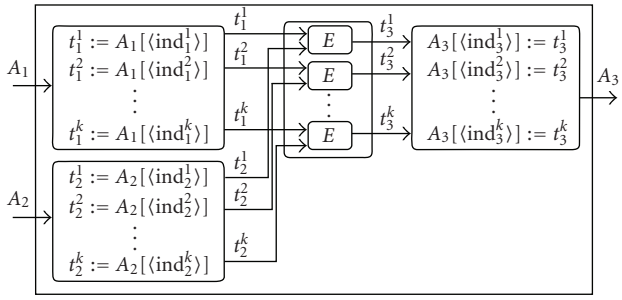


Figure 8: Modeling of tilers and parallel tasks.

the execution model of GASPARD. Therefore, we concretely enforce this representation by modeling Array-OL arrays into flows of subarrays.

(r4) The body is translated according to the appropriate rule, being either an elementary task (r5), a repetitive task (r6), or a hierarchical task (r9).

(r5) An elementary task E is represented by one equation, a Signal *relation* construct, defining the output in terms of a function of the input, essentially of the form: $O := E(I)$.

This equation can be encapsulated in an instantaneous process P_E (with bounded execution duration), which has the same interface as E. All input parameters of P_E precede all its output parameters.

(r6) Repetitive tasks are modeled by a compound Signal node, where

(i) the input tilers are transformed according to (r8),

(ii) the repeated computation is represented by a node with a composition of $|\mathbf{r}|$ instances of the node representing the repeated body, obtained by (r1),

(iii) the output tilers are transformed according to (r8).

Figure 8, showing the repetition of body E, graphically illustrates the perfect matching of this Signal compound node with the GASPARD repetitive task of Figure 2.

(r7) Connexions are translated as assignments between the p_i and p_0 ports.

(r8) Each tiler is represented by a node; for input tilers, this node takes as input the array given by the connection in (r7), and where each pattern is produced as output, by an equation extracting it from the input array. The indexes for each pattern are obtained by applying the paving and fitting equations, as explained below. Output tilers are represented by a node, where each pattern, given by the connection in (r7), is inserted in the output array by an equation.

(r9) Hierarchical tasks are modeled by the synchronous composition of nodes representing each of the subtasks, obtained by (r1), with the appropriate data dependencies defined by the connections in (r7).

The hierarchy of GASPARD task models is trivially described using the process hierarchy of Signal: if a task H at a given level is modeled by a process P, the subtasks associated with H will be modeled by subprocesses of P. Indeed, both hierarchy representations are of the same nature. The example presented in Section 2.3.3 illustrates this correspondence between the GASPARD model depicted by Figure 9 and its Signal model given in Figure 10.

2.3.2. *Correctness with respect to the GASPARD semantics*

The correctness of the transformation with respect to the sets of computations defining the behavior of GASPARD can be examined essentially at three levels as follows.

(i) An elementary task is transformed in Signal as an individual invocation of the corresponding function, hence one computation in GASPARD is trivially transformed into one corresponding computation in Signal.

(ii) Hierarchical tasks, defining the union of underlying sets of computations, are transformed into a synchronous composition of Signal nodes representing the subtasks, hence the resulting system of equations is the union of the corresponding sets of equations, hence the resulting set of computations is simply the union of the underlying sets of computations.

(iii) Repetitive tasks are not directly transformed into sets of basic computations, so we have to derive this basic form from the structured compound node proposed above.

The proposed transformation has the form of the following equations system (4), strictly equivalent to its graphical form in Figure 8:

$$
\begin{aligned}
&(\,|\,t_1^1 := A_1[\langle \text{ind}_1^1 \rangle]\,|\cdots|\,t_1^k := A_1[\langle \text{ind}_1^k \rangle]\\
&\;|\,t_2^1 := A_2[\langle \text{ind}_2^1 \rangle]\,|\cdots|\,t_2^k := A_2[\langle \text{ind}_2^k \rangle]\\
&\;|\,t_3^1 := E(t_1^1, t_2^1)\,|\cdots|\,t_3^k := E(t_1^k, t_2^k)\\
&\;|\,A_3[\langle \text{ind}_3^1 \rangle] := t_3^1\,|\cdots|\,A_3[\langle \text{ind}_3^k \rangle] := t_3^k\\
&\;|\,)\quad \textit{where } t_1^1, t_2^1, t_3^1, \ldots, t_1^k, t_2^k, t_3^k \textit{ end},
\end{aligned}
\tag{4}
$$

where in the two first lines, patterns are extracted from the two input arrays, in the third line, the k instances of the repeated task E are applied, and the fourth line shows how patterns are recomposed into the output array.

Indexes $\langle \text{ind}_i^j \rangle$ denote the indexes associated with the elements of the tile corresponding to the point j in the repetition space; and E is the synchronous model of the computation part, corresponding to an elementary task. Here, a partial assignment of arrays is assumed for the sake of

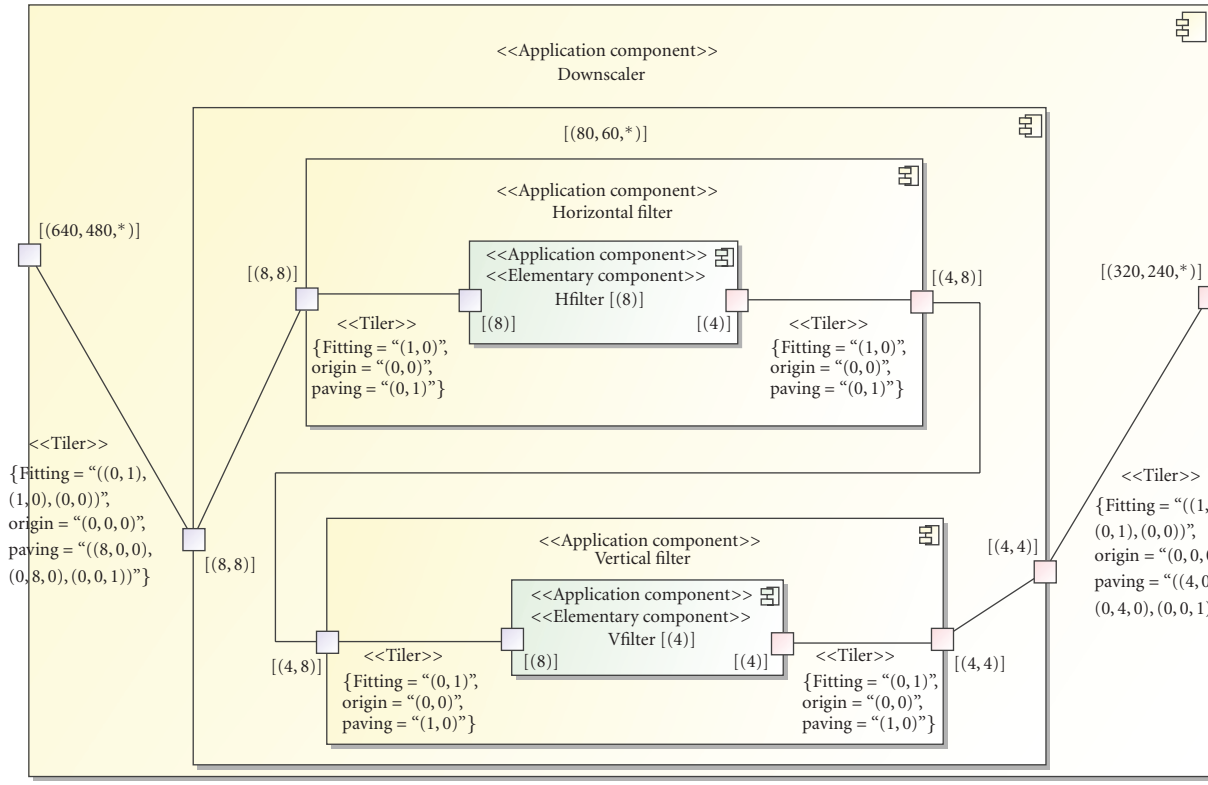


Figure 9: A model of downscaler according to GASPARD.

clarity. If the tiler for an array A_i is $(F_i, \mathbf{o}_i, P_i)$, $\mathbf{s}_{\mathbf{p}_i}$ is the shape of the pattern associated with tiles in array A_i and $\mathbf{s}_{\mathbf{a}_i}$ is the shape of array A_i, then $\langle \text{ind}_i^j \rangle$ is the following set of indexes:

$$\langle \text{ind}_i^j \rangle = \{\mathbf{o}_i + jP_i + \mathbf{x}F_i \bmod \mathbf{s}_{\mathbf{a}_i}, \text{ where } \mathbf{0} \le \mathbf{x} < \mathbf{s}_{\mathbf{p}_i}\}. \quad (5)$$

Observe that this system (4) does not present a clear separation of single repetition points. Actually, such a separation can be restored by simply considering the commutativity and associativity properties of the Signal composition operator. By just permuting equations, the following system (6) is straightforwardly deduced from system (4):

$$\begin{aligned}
&(\,|\, t_1^1 := A_1[\langle \text{ind}_1^1 \rangle] \,|\, t_2^1 := A_2[\langle \text{ind}_2^1 \rangle] \\
&\;|\, t_3^1 := E(t_1^1, t_2^1) \,|\, A_3[\langle \text{ind}_3^1 \rangle] := t_3^1 \\
&\;|\, \cdots \\
&\;|\, t_1^k := A_1[\langle \text{ind}_1^k \rangle] \,|\, t_2^k := A_2[\langle \text{ind}_2^k \rangle] \\
&\;|\, t_3^k := E(t_1^k, t_2^k) \,|\, A_3[\langle \text{ind}_3^k \rangle] := t_3^k \\
&\,|\,) \quad \textit{where } t_1^1, t_2^1, t_3^1, \ldots, t_1^k, t_2^k, t_3^k \textit{ end};
\end{aligned} \quad (6)$$

where each line describes the treatment of one point j in the repetition space.

Each line of system (6) consists of the introduction of a few intermediary local signals in (7), hence it can be trivially proved equivalent:

$$\begin{aligned}
&\forall j \in \text{repetition space}, \\
&A_3[\langle \text{ind}_3^j \rangle] := E(A_1[\langle \text{ind}_1^j \rangle], A_2[\langle \text{ind}_2^j \rangle]).
\end{aligned} \quad (7)$$

Indeed, it is costless to introduce intermediary signals, as they can be compiled away by the synchronous compilers and optimizers.

The modeling of a repeated computation then amounts to the synchronous composition of all the models of each point in the repetition space:

$$\begin{aligned}
&(\,|\, A_3[\langle \text{ind}_3^1 \rangle] := E(A_1[\langle \text{ind}_1^1 \rangle], A_2[\langle \text{ind}_2^1 \rangle]) \\
&\;|\, \cdots \\
&\;|\, A_3[\langle \text{ind}_3^k \rangle] := E(A_1[\langle \text{ind}_1^k \rangle], A_2[\langle \text{ind}_2^k \rangle]) \\
&\,|\,),
\end{aligned} \quad (8)$$

where k denotes the number of repetitions. As mentioned previously, Array-OL expresses only data dependencies between arrays, hence leaving all the potential parallelism in the specifications. In particular, repetitions on arrays describe how many times, and according to what paving and fitting a computation should be repeated, but *not* in what order the array elements should be accessed. In other words, *any order* could be adopted in an iteration implementing such a repetition in the case of a sequential implementation provided that functional determinism is preserved. We conform to this property of Array-OL simply by using synchronous composition between models of all the repetitions, thereby inducing no order between them.

Hence, the synchronous model of repetitive tasks describes the same set of computations as the GASPARD repetitive task.

```
module Downscaler_module =
%External libraries%
  use Basic_Functions;
%Types and constants%
  type type_array_i = [640,480] integer;
  type type_array_o = [320,240] integer;
  type type_tile_i = [8,8] integer;
  ...

 process DOWNSCALER =
  (? type_array_i A_i;
   ! type_array_o A_o;)
  (| (T_i1,...,T_i4800) := HV_TILER_i(A_i)
   | (T_o1,...,T_o4800) :=
       R_HV_FILTER(T_i1,...,T_i4800)
   | A_o := HV_TILER_o(T_o1,...,T_o4800)
   |)
  where
   type_tile_i T_i1,...,T_i4800;
   type_tile_o T_o1,...,T_o4800;
   process HV_TILER_i =
   (? type_array_i A_i;
    ! type_tile_i T_i1,...,T_i4800;)
   (| T_i1 := [[A_i[0,0],...,A_i[0,7]],
              ...,
              [A_i[7,0],...,A_i[7,7]]]
    |...
    | T_i4800 :=
     [[A_i[631, 471],...,A_i[631, 479]],
      ...,
      [A_i[639, 471],...,A_i[639, 479]]]
    |)
    end; %process HV_TILER_i%
   process R_HV_FILTER =
   (? type_tile_i T_i1,...,T_i4800;
    ! type_tile_o T_o1,...,T_o4800;)
   (| T_o1 := HV_FILTER(T_i1)
    |...
    | T_o4800 := HV_FILTER(T_i4800)
   |)
   where
    process HV_FILTER =
    (? type_tile_i T_i;
     ! type_tile_o T_o;)
    (| t := H_FILTER(T_i)
     | T_o := V_FILTER(t)
     |)
     where
      type_tile_l t;
      process H_FILTER =
      (? type_tile_i T_i;
       ! type_tile_l t;)
      (|...|)
      process V_FILTER =
      (? type_tile_l t;
       ! type_tile_o T_o;)
      (|...|)
      end; %process HV_FILTER%
    end; %process R_HV_FILTER%
  process HV_TILER_o =
   (? type_tile_o T_o1,...,T_o4800;
    ! type_array_o A_o;)
   (| A_o := HV_TILE_o(T_o1,...,T_o4800)
    |)
    where
    process HV_TILE_o =
    (? type_tile_o T_o1,...,T_o4800;
     ! type_array_o A_o;)
    (| A_o :=
      [[T_o1[0,0],...,T_o4720[0,3]],
      ...,
      [T_o80[0,0],...,T_o4800[3,3]]]
    |)
    end; %process HV_TILER_o%
  end; %process DOWNSCALER%
end; %module Downscaler_module%
```

Figure 10: A sketch of the signal code of the downscaler.

At this stage, it is possible to have a more compact synchronous notation of a repetitive task. For instance, a *map* of the function E on the array [39] or an *array of processes* construct [13] offer this possibility. The model presented here is meant to be intuitive and simple, and may certainly be optimized. However, the considered optimizations can be quite different according to the intended target operations (e.g., efficient code generation or verification).

2.3.3. Application to the description of an image downscaler

As an example to illustrate our modeling approach, let us consider the video functionality of a new generation multimedia cell phone. Such mobile devices are being increasingly adopted by consumers. They always include multimedia features, for example, camera, MP3 player, or radio provided by multimedia modules and processors integrated in chips. These features make the system design more complicated than ever, leading to real development challenges today in telephony industries.

The part of the video functionality modeled here deals with scaling. It consists of a classical downscaler, which transforms a video graphics array signal (VGA), 640×480 pixels per frame into a quarter video graphics array (QVGA) signal, 320×240 pixels per frame. Therefore, a downscaling of 4:1 is required. Such an operation is interesting when visualizing high quality live video in thin-film transistor (TFT) screen while using low power and real-time previews (i.e., *view* mode in video functionality of a cell phone) TFT refers to active matrix screens on laptop computers, which offers sharper screen displays and broader viewing angles than does passive matrix. The best-known application of TFT is in liquid crystal display (LCD) technology. The downscaler itself is composed of two components: a horizontal filter that reduces the number of pixels from a 640-line to a 320-line by interpolating packets of 8 pixels; and a vertical filter that reduces the number of pixels from a 480-line to a 240-line by interpolating packets of 8 pixels as well.

GASPARD model

The GASPARD model of the downscaler is illustrated by Figure 9. It is represented by a hierarchical task consisting of three levels: top level, a repetitive task referred to as *Downscaler*; second level, a compound component

represented by a directed acyclic graph where the nodes are repetitive tasks *Horizontal filter* and *Vertical filter*; and third level, elementary tasks *Hfilter* and *Vfilter* that are repeated within the repetitive tasks of second level. The whole downscaler receives an infinity of frames, denoted by the input 3D array $(640, 480, \infty)$, and produces an infinity of transformed frames, $(320, 240, \infty)$. Here, ∞ is represented by the symbol $\star$ in the depicted model. The way pixels are extracted from (*resp.,* inserted in) these infinite arrays is described by tilers.

Signal description

The Signal description sketched in Figure 10 corresponds to the code generated automatically from the GASPARD model of the downscaler by our implemented transformation. The whole application is encapsulated in a *module* called *Downscaler_module*. A module plays a similar role as a package in UML. Other modules are imported via the statement *use*: *Basic_Functions* proposes elementary external functions, such as FFT, DCT, which are performed by elementary tasks. In addition, types and constants can be declared in the module. For instance, *type_array_i* is defined as an array of integer type with size [640, 480].

The main process in the module, called *DOWNSCALER*, reflects the same hierarchy as the corresponding GASPARD model. Here, due to lack of space, we only represent the main parts of process *DOWNSCALER*. This process is composed of three subprocesses: *HV_TILER_i* for input tiler, *R_HV_FILTER* containing the task repeated at the top level, and *HV_TILER_o* for output tiler. Let us focus first on the definition of *HV_TILER_i* in the Signal code. It takes the input array *A_i* and produces 4800 tiles, each associated with a point in the repetition space. In particular, we can notice that index values in arrays are static since they are determined during the transformation from the GASPARD model to the synchronous descriptions. The process *R_HV_FILTER* represents the top-level repeated task, and contains two subprocesses denoting the compound tasks in the second level of the global hierarchy: *H_FILTER* and *V_FILTER*. Their output tiles are stored in *A_o* by process *HV_FILTER_o*.

Observations

The resulting synchronous description of the downscaler perfectly matches the corresponding GASPARD model. The advantage is that it can be considered for an analysis using the formal tools and techniques available in the Signal environment. This is addressed in Section 3. However, the main limitation of such a model concerns scalability. The size of the process sketched in Figure 10 linearly depends on the number of repetitions performed in parallel (here, 4800 subsets of equations). A possible way to reduce the size of the process consists in considering that some computations, for example, tile index calculations, are achieved by external modules. But, such a solution is only partially satisfactory.

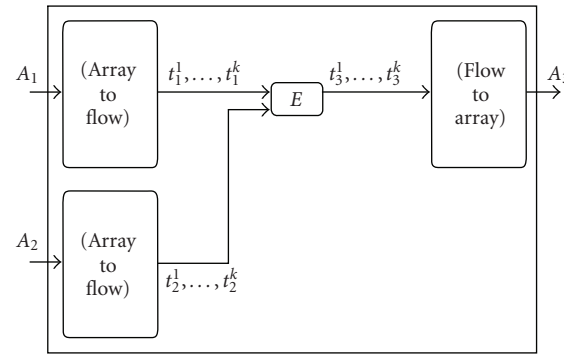


Figure 11: A serialized model.

2.4. Serialized model

2.4.1. Structural transformation

Avoiding enumerating repetitions

While the parallel model fully preserves the data parallelism present in Array-OL specifications, the model called *serialized*, which is described in this section offers a refined view of such specifications by sequentializing the execution of a repetitive task. This model is particularly compact in comparison with the parallel model because it only defines one instance of the repeated task in a repetitive task. It therefore significantly reduces the scalability problem mentioned before. It also features a monoprocessor execution of multiple repetitions. More generally, GASPARD applications will be modeled by combining both parallel and serialized models (Section 2.5). The serialized version of elementary tasks and of hierarchical tasks do not differ from their parallel version from the viewpoint of model construction. Here, we mainly focus on the translation of repetitive tasks.

An alternative transformation

The difference with the previous transformation mainly concerns the rules describing repetitive tasks (r6) and the tilers (r8), which is as follows.

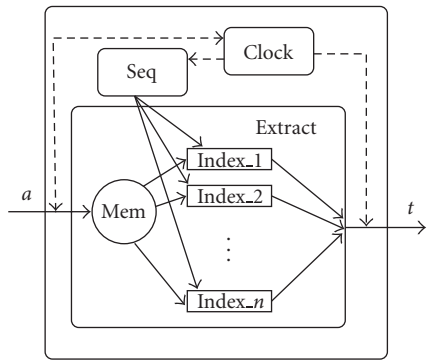
(r6′) Repetitive tasks are modeled by a compound Signal node, where

(i) the input tilers are transformed according to (r8′);

(ii) the repeated computation is represented by a single instance of the node representing the repeated body, obtained by (r1);

(iii) the output tilers are transformed according to (r8′).

As shown by Figure 11 in our graphical syntax for Signal, in other words, we distinguish three basic subparts: a single task instance E (which can still be seen as an external function) and two kinds of components referred to as *Array to Flow* and *Flow to Array*. E receives its input tiles via a tile flow from *Array to Flow* and sends its output tiles to *Flow to Array*, also via a tile flow of the same length as the one given by *Array to Flow*.




```
(1) process Array2Flow = { integer k; type array_type, tile_type; }
(2)   (? array_type a; ! tile_type t; )
(3)   (| clk_p := Clock{k} (a)
(4)    | id := Seq(clk_t)
(5)    |t := Extract{array_type, tile_type}(id, a)
(6)    |)
(7)    where
(8)     event clk_t, integer id;
(9)     process Extract = {type array_type, tile_type; }
(10)      (? integer id; array_type a; ! tile_type t;);
(11)     process Seq =
(12)      (? event c ! integer id);
(13)     process Clock = {integer k; }
(14)      (? event c1; ! event c2; )
(15) end; %process Array2Flow%
```

Figure 12: Construction of tile flows from arrays.

(r8′) Each tiler is represented by a node defined by the *Array to Flow* and *Flow to Array* components, which play a central role in the serialized model. This definition partially relies on the oversampling mechanism introduced previously.

Let us consider that incoming arrays A_1 and A_2 are received at the instants $\{\tau_{i,i\in\mathbb{N}}\}$ of a given clock c_1. Then, for each τ_i, the tile production algorithm is applied to the received arrays: the task instance E is provided with the flow of tile pairs (t_1^j, t_2^j) and produces the tile flow t_3^j; the resulting tile flow is therefore associated with a clock $c_2 = k \uparrow c_1$, where $0 \le j \le k$. The constant integer k corresponds to the number of paving iterations deduced from the repetition space. It is directly derived from Array-OL specifications.

Globally, the definition of *Array to Flow* and *Flow to Array* components includes the following aspects.

(1) *Memorization*: in *Array to Flow*, every incoming array must be made available at every instant of c_2 in order to extract the successive output tiles. Similarly, in *Flow to Array*, the incoming tiles must be accumulated locally until the k expected tiles become available. Then, the output array can be produced.

(2) *Consumption and production rates*: in both components, we describe the frequencies at which tiles and arrays are consumed or produced. This is basically captured through the clock information associated with the corresponding signals.

(3) *Scheduling*: we have to ensure the coherency between the tile flow produced by *Array to Flow* and the one consumed by *Flow to Array*. Here, we consider a common sequencer for both components, which decides the right tiles to be scheduled at every instant τ_i of the clock associated with a tile flow.

Construction of tile flows from arrays

The *Array to Flow* component is described in both graphical and Signal textual syntax by Figure 12. The illustration given on the left-hand side is encoded by the Signal program shown on the right-hand side (we do not give the detailed Signal program due to lack of space).

Three parts are to be distinguished. First, the *Clock* subprocess (line 3 in the program) defines the tile production rate from the input array denoted by *a*. It basically consists of the oversampling mechanism specified previously (i.e., process *k_Overspl*). Then, the *Extract* subprocess (line 5) is used to memorize an incoming array from which it extracts the tiles *t*. Finally, the *Seq* subprocess (line 4) describes in which order tiles are gathered from the input array. It produces tile identifiers *id* at the same rate as the event *clk_t* defined by *Clock*. These identifiers are used by the *Extract* subprocess to produce tiles. The tile enumeration strategy adopted in *Seq* can be decided from several viewpoints as follows.

(i) *Environment*: the presence of sensors/actuators imposes particular orders according to which data are received/produced.

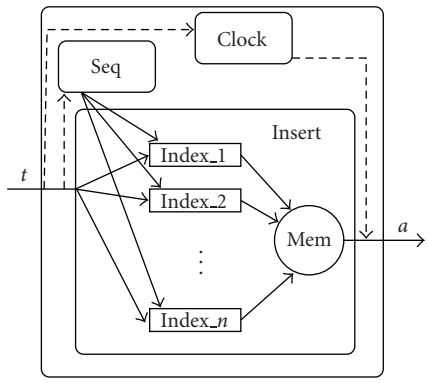
(ii) *Application*: in some algorithms, each computation step requires the results obtained at previous steps, referred to as *interrepetition dependency* in Array-OL [5], hence leading to precedence constraints between task instances. Operating modes in some applications are also source of flows [35].

(iii) *Architecture*: characteristics related to architecture devices such as processor or memory may also lead to particular scheduling/allocation strategies.

Reconstruction of arrays from tile flows

The description of the *Flow to Array* component (see Figure 13) is obtained in a symmetric way as *Array to Flow*.

Here, in the *Clock* subprocess, there is no need to use the oversampling mechanism. However, we have to define the instant when the output array is produced. This is represented by the signal *clk_a*, which occurs whenever k input tiles are received. Every input tile is immediately stored in the memorized local array tmp within the $Insert$ subprocess. The *Seq* subprocess is the same as previously. It defines the way the tiles are organized within the local array by indicating the suitable index (*id*) associated with them.




```
(1) process Flow2Array = { integer k; type array_type, tile_type; }
(2)   (? tile_type t; ! array_type a; )
(3)   (| clk_a := Clock{k} (^t)
(4)    | id := Seq (^t)
(5)    | tmp := Insert { array_type, tile_type } (id, t)
(6)    | a := tmp when clk_a
(7)    |)
(8)   where
(9)      integer id; event clk_a; array_type tmp;
(10)     process Insert = {type array_type, tile_type; }
(11)      (? integer id; tile_type t; ! array_type a; );
(12)     process Seq =
(13)      (? event c ! integer id );
(14)     process Clock = {integer k; }
(15)      (? event c1; ! event c2; )
(16) end; %process Flow2Array%
```

Figure 13: Construction of arrays from a tile flow.

2.4.2. Correctness with respect to the GASPARD semantics

The correction of the serialized model is based on the fact that the set of computations performed is that of the GASPARD definition. Indeed,

(i) the oversampling is done in such a way that the repeated function will be invoked as many times as the size of the repetition space;

(ii) each invocation will apply on an input pattern value obtained from the input array by using the appropriate index obtained from the tiler definition;

(iii) each produced output pattern value is stored in the output array by using the appropriate index obtained from the tiler definition;

(iv) the fact that the same sequence is applied in *Array to Flow* and *Flow to Array* ensures that indexes for input and output correspond to the same point in the repetition space;

(v) the input and output arrays of the repetitive tasks hence contain the same values as in the previous case where they were computed in parallel; the fact that they are serialized, along the oversampled local clock, is actually completely internal to the Signal process. Seen from the external point of view, the input array will be transformed into an output array, not in the same logical instant, as was the case previously, but in a later instant. However, the flows of input arrays and output arrays will carry the same values in the same order.

The parallel and serialized models offer the bases to represent GASPARD applications from high-level views (i.e., with a maximal parallelism) to more refined views (i.e., with serialized execution). We have a strict functional equivalence (or flow-equivalence [7]) between the parallel and serialized models: the input and output arrays in both cases are the same although the order in which their constituent elements are produced may differ. For any two repetitions, there is no side effect between their execution. So computed patterns always hold the same value independently from the kind of model.

2.5. Mixing parallel and serialized models

We now present how the parallel and serialized models can be combined in order to describe an application composed of different kinds of tasks. The model resulting from such a combination typically features the projection of the application on a multiprocessor architecture (see the methodology illustrated in Figure 1), where each serialized model represents a monoprocessor execution of a subpart of the application.

Composition

The composition of several parallel synchronous models of elementary tasks is quite straightforward. Indeed, the execution of the resulting description is instantaneous. The parallel composition of these tasks is exactly the same as the one of processes in synchronous languages.

When serialized models are involved, the parallel composition is more tricky because of multiple oversamplings within composed elementary tasks. To illustrate the problem, let us consider two serialized tasks T_1 and T_2 with different oversampling parameters, such that their outputs are the only inputs of a task T_3. According to the execution model of Array-OL defined in GASPARD, T_3 imposes the simultaneous presence of its inputs before executing. To ensure this property, we have to make the outputs of T_1 and T_2 available at the same moment although they have different oversampling rhythms (since one task will probably finish its execution before the other).

The process given in Figure 14 describes our solution, which synchronizes k inputs of a task. Each array is associated with a state variable represented by Boolean c. The outputs are produced when the signal rdv (the conjunction or all state variables) is *true*. Here, the Signal construct $y := x$ *cell* b means that y takes the value of x when x is present, otherwise the latest value of x whenever the Boolean b holds the value *true*.

Hierarchy

The hierarchy of GASPARD models can be represented in synchronous languages by considering their modularity.

```
(1) process synchronize = {type array_type; }
(2)   (? array_type a1,...,ak; ! array_type a11,...,akk; )
(3)   (| (| c1^= ··· ^= ck^= (a1 ^+ ··· ^+ ak)
(4)      | c1 := ^a1 default (false when (rdv$1)) default (c1$1 init false)
(5)      | ···
(6)      | ck := ^ak default (false when (rdv$1)) default (ck$1 init false)
(7)      | rdv := c1 and ··· and ck
(8)      |)
(9)   | (| a11 := (a1 cell rdv) when rdv
(10)     | ···
(11)     | akk := (ak cell rdv) when rdv
(12)     |)
(13)  |)
(14)  where boolean c1,...,c2, rdv;
(15) end; %process synchronize%
```

Figure 14: A synchronize in signal.

Every nonhierarchical task of GASPARD is replaced by either a parallel or a serialized model depending on the execution. Then, for each hierarchical task at level n, represented by a process in Signal, its subtasks are considered as subprocesses of the synchronous model associated with the task at level n (see Figure 10).

3. ANALYSIS OF GASPARD MODELS USING SYNCHRONOUS TOOLS

We present how the synchronous models obtained from the previous section are used to address design correctness issues for GASPARD models. We primarily discuss how basic characteristics of GASPARD models can be easily checked via the corresponding synchronous models (Section 3.1). Then, we present two specific analyses on these models. The first analysis concerns causality analysis for GASPARD models (Section 3.2). It consists in checking that the specified data dependencies do not induce any cycle in the global graph representing a model. The second analysis specifically addresses nonfunctional properties (Section 3.3). It deals with constraints on execution frequencies in systems designed with GASPARD. One may need to guarantee that the application implementation respects some given data production rates at the modeling phase in order to reduce the global design cost earlier. This knowledge has an impact on the choice of the granularity of array processing. Here, we propose to address this issue by using specific concepts of synchronous languages though a simple but relevant example. Further possible analyses of GASPARD models are also briefly mentioned (Section 3.4). To finish, some implementation aspects are discussed (Section 3.5).

3.1. Guaranteeing single assignment and functional determinism

We discuss how the translation described in this paper makes it possible to check some basic properties of GASPARD models with the synchronous technology.

3.1.1. Single assignment

In Section 1.2, we have seen that single assignment is one of the key characteristics of Array-OL. Thus, for a given output array A of a task, one must ensure that no element of A, denoted by $A\,[\langle \text{ind} \rangle]$, is overwritten after its first value assignment. This typically happens when the paving matrix and the shape of patterns lead to tiles that overlap within A. Single assignment in a GASPARD model can be checked with an algorithm that exhibits the emptiness of polyhedra intersection as shown in [5]. Such an algorithm can be implemented using linear programming. However, no implementation is currently available in the GASPARD environment.

The synchronous models resulting from our translation can therefore be used to check single assignment in the translated GASPARD models. As a matter of fact, most of the considered synchronous dataflow languages assume single assignment. It is the case of Lustre and Signal. Their associated compilers therefore allow one to check this property on programs.

Here is an example that illustrates the single assignment issue. Let us consider the rightmost output tiler of the *Downscaler* model illustrated previously in Figure 9. The left-hand side picture in Figure 15 (taken from the Array-OL tool - (http://www2.lifl.fr/west/aoltools)) partially shows the correct paving according to the tiling information defined in Figure 9. Patterns have the shape (4,4) and are separated by white lines. They are perfectly contiguous. In each pattern, an array element with white circles around denotes the reference element of the pattern.

It is not the case for the tiling information considered for the picture on the right-hand side in Figure 15. These information are as follows:

$$F = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad o = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \qquad P = \begin{pmatrix} 3 & 0 \\ 0 & 4 \end{pmatrix}. \tag{9}$$

We can observe intersection regions between some contiguous tiles in the picture. For instance, pattern $P_{(0,0)}$ overlaps pattern $P_{(1,0)}$ at the array elements with indexes $(3,0)$, $(3,1)$, $(3,2)$, and $(3,3)$.

The translation of a GASPARD model with a tiling information as illustrated in the last picture will produce a synchronous description with multiple assignments to the same variable, thus violates the single assignment property required by Array-OL.

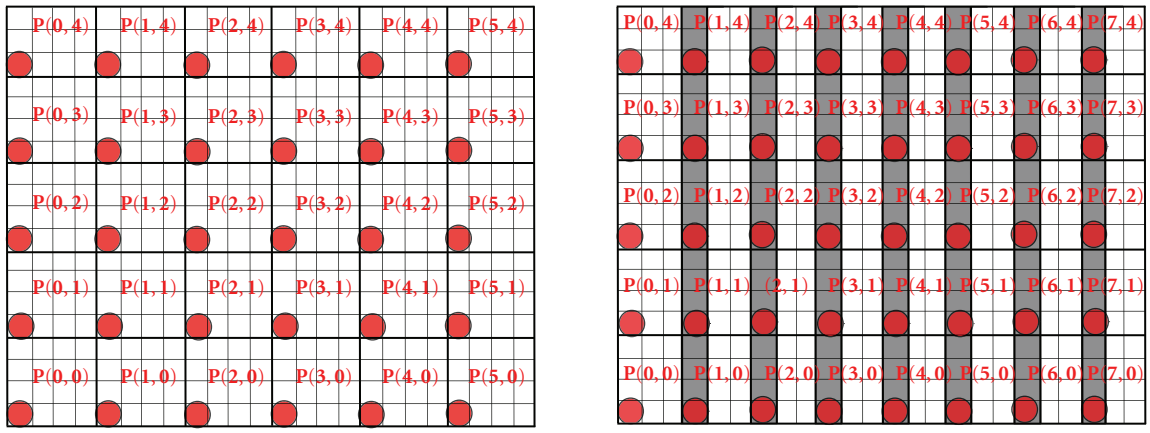


Figure 15: The illustrations of the single assignment issue.

3.1.2. Functional determinism

Synchronous languages have been originally introduced in order to enable deterministic descriptions of embedded system behaviors. Given a synchronous model, its functional determinism is trivially verified using the compiler. For instance, in Signal, the so-called *endochrony* property that is checked with the compiler ensures that a program behaves deterministically [7]. In Lustre, programs are deterministic by construction. If the code generated by our transformation is not deterministic, it will be systematically rejected by the Lustre compiler. Our transformation of GASPARD models does not a priori take into account the determinism during the code generation process. Such a property is only checked on the resulting synchronous code when required. Nevertheless, note that non deterministic behaviors could be interesting when we want typically to model system environment. For instance, the concept of sensor in GASPARD can be associated with a model that behaves nondeterministically regarding the production of values.

3.2. Causality analysis

The absence of causality cycles in GASPARD models is another important property of GASPARD models. Currently, designers are not provided with any specific tool in the GASPARD environment to check the absence of causality cycles in models. While such cycles may be avoided easily in simple application models, the detection of their presence in more complex models, for example, models with several nested hierarchical levels or models including control features [35], is very far to be trivial. As mentioned in Section 2, such a property of Array-OL specifications could be addressed with polyhedra techniques (e.g., [34]). However, for the GASPARD extension of Array-OL, we also argued why these techniques become less attractive compared to those relying on mixed representations like the Signal HCDG.

Further situations where causality analysis is interesting in GASPARD concern IP reuse in the associated design methodology (see Figure 1). As a matter of fact, the use of some IPs during the deployment phase may introduce data-dependency cycles in cycle-free models that result from the association phase. Conversely, some IPs can break an apparent data-dependency cycle in a model (see the examples shown in Figure 17). Our translation of GASPARD models toward synchronous languages offers an efficient way to deal with all these issues with their corresponding compilers. The presence of a cycle in a synchronous program leads to deadlock, that is, instantaneous self dependency.

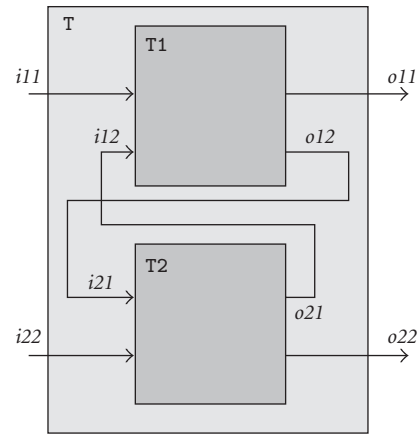
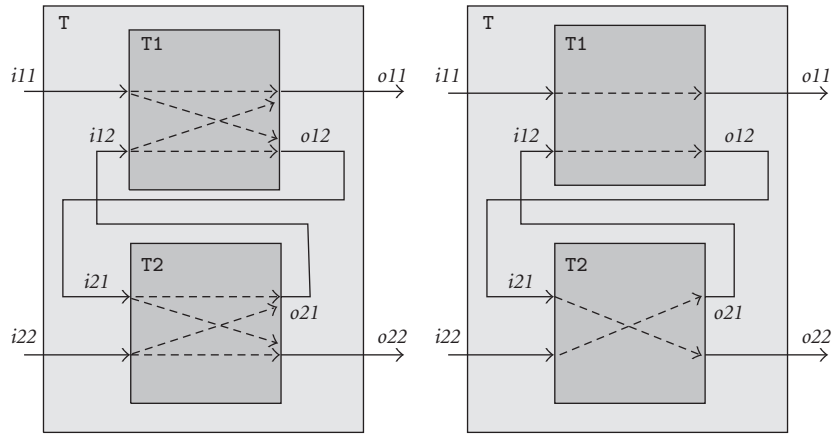


Figure 16: A simple hierarchical task model.

Let us consider the hierarchical task T, depicted by Figure 16. This task is composed of two subtasks $T1$ and $T2$, which communicate with each other via some local array variables. For the sake of simplicity, these subtasks are assumed to be elementary tasks. They are therefore represented by either an IP or some black-box abstraction of their corresponding functionality. Now, we want to check whether or not there is a causality cycle in the specification of T. Depending on the level of detail of a translated GASPARD model into a synchronous model, the cycle detection can be achieved either mainly syntactically as in the Lustre language [40]; or using more sophisticated techniques as in the Signal



(a) Detection of causality cycle in T (b) Absence of causality cycle in T

Figure 17: Causality analysis for GASPARD application models at different levels of detail.

language which takes into account clock information to decide the validity of data dependencies [41].

3.2.1. According to the execution model defined in GASPARD

Figure 17(a) illustrates the task T according to the execution model of Array-OL defined in GASPARD, every output port of an elementary task (e.g., $T1$ or $T2$), say $o11$ in $T1$, depends on all input ports of the task, that is, $i11$ and $i12$. The translation of such an Array-OL model leads to a synchronous program on which a dedicated compiler can straightforwardly exhibit the presence of causality cycles. As a result, the version of task T given in Figure 17(a) should be rejected with respect to the current semantics of elementary tasks in Array-OL.

3.2.2. According to a finer grained execution model

Now, let us consider that a user has several alternatives concerning the way elementary tasks $T1$ and $T2$ can be defined. In other words, he has different IPs that implement both $T1$ and $T2$. A possible choice may be the one illustrated in Figure 17(b). Here, the dependency constraint induced by the execution model of Array-OL in GASPARD, on interface ports is no longer considered. A finer-grained description of the interfaces of the IPs is available, enriched with dependency information. For instance, in $T1$, $o11$ only depends on $i11$. Assuming the dependencies specified in Figure 17(b), it is very easy to show that the translation of the second version of task T leads to a deadlock-free program. With a finer-grained view of models, that is, a more precise interface description of tasks, the causality analysis avoids the detection of "false" cycles. As a result, a finer compilation of specifications and better reuse of IPs are made possible.

The above example shows another advantage of the synchronous modeling in that it enables to explore the presence of causality cycles in Array-OL models from different execution models of an application. This allows one to identify a wider range of valid application specifications while still respecting a basic principle of Array-OL, consisting of the absence of causality cycles in models. Note that there is a great opportunity of using conditioned dependencies, as is done in the Signal causality analysis. This is particularly very useful in GASPARD with control features [35].

	0	1	2	3	4	5	6	7	8	9	10	⋯
c_1 :	tt	⊥	⊥	tt	⊥	⊥	tt	⊥	⊥	tt	⊥	⋯
c_2 :	⊥	tt	⊥	⊥	⊥	tt	⊥	⊥	⊥	tt	⊥	⋯

Figure 18: c_1 and c_2 are in $(3, 1, 4)$-affine relation.

3.3. Synchronizability analysis using affine clocks

We first recall the definition of affine clocks. Then, we use this notion to deal with clock synchronization issues in an application example.

Affine clocks

Affine clocks enable to deal with synchronizability issues in a more relaxed way than usually in synchronous languages. They have been defined by Smarandache et al. [36] in order to address the validation of real-time systems using the functional data parallel language Alpha [22] and Signal. Intensive numerical computations are expressed in Alpha while the control (the clock constraints resulting from Alpha descriptions after transformations) is addressed in Signal. The regularity of computations expressed in Alpha makes it possible to identify affine relations between the specified clocks. The Signal compiler therefore enables to address synchronizability criteria based on such clock relations.

Definition 1. An *affine transformation* of parameters (n, ϕ, d) applied to a clock c_1 produces a clock c_2 by inserting $(n - 1)$ instants between any two successive instants of c_1, and then counting on this fictional set of instants each dth instant, starting with the ϕth. Clocks c_1 and c_2 are said to be in (n, ϕ, d)-affine relation, noted as $c_1 \xrightarrow{(n,\phi,d)} c_2$ (see example in Figure 18).

In affine clock systems, two different signals are said to be synchronizable if there is a dataflow preserving way to make them actually synchronous. The Signal *clock calculus* (i.e., its static analysis phase) has been extended in order to address such synchronizability issues with the associated compiler [36].

Using affine clocks

Let us consider a cell phone in its *View* mode of video functionality [42]. It captures video images through its CMOS sensor, and these images are immediately transferred to the processor. Then, in general, the images should be downscaled so as to fit for the TFT preview. Finally, the downscaled images must be transferred to TFT for display. To simplify the modeling of this functionality, we assume that there exists a base clock in this cell phone. The sensor, the processor and TFT display hold a logical clock based on the base clock through affine relations, which also lead to further affine relations between the logical clocks characterizing the three components. Now, we want to analyze how these components must be synchronized through their clocks so that the video can be normally displayed in the TFT. This issue is addressed below by performing a synchronizability analysis on the resulting synchronous model by using affine clocks.

In Figure 19, the downscaler is considered together with its connected components in the cell phone. The way it receives its inputs and outputs will be now expressed through flows. A CMOS sensor gathers data and sends to the downscaler a flow of pixels, denoted by t_i^k. Then, the downscaler transforms these pixels in order to display reduced video images on the TFT screen. The result of this transformation is the flow t_o^k.

Each component is associated with a logical clock that defines its activation instants, that is, its data consumption/production rates. This is typically what GASPARD is not able to describe. Remember that in GASPARD the only kind of relation that can be specified between different tasks is data dependency and repetition. There is no information on the presence of either flows or frequencies. For this reason, the synchronous model is primarily important here because it includes such nonfunctional information. Moreover, its associated validation techniques enable to verify very interesting properties of the modeled system.

Let us denote by c_p, c_a, and c_i the respective logical clocks of the sensor, the downscaler, and the display. They, respectively, represent the pixel production rate in the sensor, the bloc computation frequency within the downscaler, and the image production rate on the display. The whole model works as follows: the sensor produces its output data pixel by pixel; the downscaler periodically performs an operation whenever it receives from the sensor a fixed number of pixels; and the TFT screen periodically displays an image whenever it receives from the downscaler a fixed number of blocs of transformed pixels. A step in c_p, c_a, and c_i corresponds to the production of, respectively, a single pixel by the sensor, a transformed block of pixels by the downscaler, and an image by the TFT display. From the point of view of GASPARD,

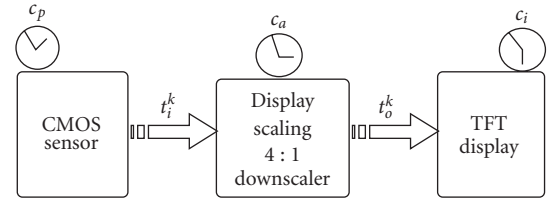


Figure 19: Downscaling images within a cell phone.

the clock steps associated with a component correspond to its paving iterations. We therefore derive the following constraints between above logical clocks:

(i) C_1 : c_a is an affine undersampling of c_p, that is, $c_p \xrightarrow{(1,\phi_1,d_1)} c_a$;

(ii) C_2 : c_i is an affine undersampling of c_a, that is, $c_a \xrightarrow{(1,\phi_2,d_2)} c_i$;

Now, let us consider a specification requirement of the video display functionality, consisting of a constraint on the actual production rate, noted c_i', of displayed images in the cell phone. This constraint, denoted by C_3, states a relation between the pixel production rate c_p and c_i' as follows: $c_p \xrightarrow{(1,\phi_3,d_3)} c_i'$. Then, we need to guarantee the compatibility of this new constraint with the previous set of constraints $\{C_1, C_2\}$. In other words, we want to establish a synchronizability relation between clocks c_i' and c_i with respect to $\{C_1, C_2, C_3\}$. This will ensure that the expected rate of the TFT display c_i, which depends on the production rate of the downscaling process c_a, satisfies the considered requirement.

This issue cannot be addressed by only using the usual definition of clock synchronization in synchronous languages. Instead, we consider affine clock systems in order to define under which condition c_i' and c_i are synchronizable. Hence, from C_1, C_2, and C_3, this synchronizability property is checked by using the following property, which has been proved in [43], and now implemented in the Signal compiler:

$$c_i' \text{ and } c_i \text{ are synchronizable} \iff \begin{cases} \phi_1 + d_1\phi_2 = \phi_3, \\ d_1 d_2 = d_3. \end{cases} \tag{10}$$

This issue is solved quite easily with synchronous models, while it is not possible with GASPARD only. The result of this analysis can be used to adjust the paving iteration parameters of the GASPARD model of the downscaler so as to satisfy the nonfunctional requirements imposed on the whole system.

In [36], Smarandache et al. combine the Alpha language and the synchronous language Signal to design and validate embedded systems by defining affine clocks relations, which are used in Section 3.3 to check a synchronizability criteria. Our approach differs from this work in that we propose a synchronous model of a whole data-parallel application instead of describing only its clock information as it is the case with [36]. As a result, our model allows us to address both functional and nonfunctional properties of the application using the synchronous technology. Another similar

work concerns the design of *n-synchronous Kahn networks* [37] in which authors consider the Lucid Synchrone language in order to address the correct-by-construction development of high performance stream-processing applications. This work also defines clock synchronizability properties that can be applicable to GASPARD models specified in Lucid Synchrone. Note that in both [36, 37], the analysis relies on clocks, which give a qualitative view of time. This is not the case of [44], which is another interesting work where authors use linear relations to analyze synchronous programs so as to verify quantitative time properties. Such an approach would be very helpful when dealing with time durations and the perspective of generating Lustre code would make the connection of GASPARD and this technique possible. For the moment, the qualitative approach of Smarandache et al. is privileged because it is more appropriate for clock synchronizability issues.

3.4. Other analyses

Among other techniques that are very useful to GASPARD applications, we mention *performance evaluation* for temporal validation. In Signal, a technique has been implemented within its associated design environment, which allows to compute temporal information corresponding to execution times [45]. It first consists in deriving from a given program another Signal program, termed the "temporal interpretation," which computes timestamps associated with the variables of the initial one. The cosimulation of both programs therefore produces at the same time the value of each variable that is present and its current timestamp (or availability date). These timing information are used to compute different latencies allowing one to calculate an approximation of the program execution time. Such an evaluation technique becomes very desirable for GASPARD in order to make architecture exploration possible early during the design flow.

We also mention the possibility to observe the functional behavior of given GASPARD applications. This is achieved by using the simulation code automatically generated by synchronous tools from the associated models. Several examples have been experimented among which are matrix and image processing. They have been implemented using the approach exposed in this paper by first defining their associated specification in GASPARD, then by generating automatically the corresponding executable synchronous code.

All these features of the synchronous technology contribute to make trustworthy the design activity for data-intensive applications in the GASPARD framework.

3.5. Implementation issues

A prototype transformation tool, based on MDE, has been developed in order to enable the automatic translation of GASPARD models into synchronous programs defined in dataflow languages [38]. It mainly relies on a generic metamodel for synchronous equational dataflow languages, which targets at the same time Lustre, Lucid Synchrone, and Signal. This tool has been developed as an Eclipse (http://www.eclipse.org/) plugin composed of a metamodel and a set of transformation rules. The implemented transformation rules globally represent about five thousands lines of Java code in Eclipse.

Figure 20 illustrates a typical transformation chain with the Signal compiler as the target technology. The data-intensive applications are first specified in MagicDraw with the help of the GASPARD UML profile. The specified model are then exported so as to be used in Eclipse environment, where automatic model transformations [38] are carried out according to GASPARD and synchronous metamodels. Then, the results of these transformations are Signal code.

The repetition size that could be handled in the transformation within the Eclipse capacity is about forty thousands. The implementations are carried out on a desktop computer that is equipped with Quad-Core Intel Xeon processors for a total of eight execution cores in two sockets. The computer has 2 gigabyte memory and runs on Linux. Eclipse environment has problems on the big memory used by its plugins, which limit the transformation plugin because it calculates array indexes and stores them in the memory temporarily. As a result, there is a maximum repetition number. But certain optimizations on memory usage can be done to improve the repetition number that can be handled, for instance, by storing the array indexes in temporary files during the computing in order to reduce the memory usage.

4. CONCLUSIONS

In this study, we propose a synchronous model for the design of data-intensive applications within the GASPARD environment, which is dedicated to high-performance system-on-chip codesign. The GASPARD underlying specification language, Array-OL, adopts a particular style where multi-dimensional arrays are manipulated. We show how models described using such a language can be modeled using synchronous dataflow languages. For that, we propose on the one hand, a synchronous model that fully preserves the data-parallelism and task parallelism of GASPARD models, and on the other hand, a refined model in which the execution of GASPARD models is partially serialized. A major advantage is that the resulting models enable to formally check the correctness of GASPARD models, which is necessary before going through the next steps of its associated design methodology (e.g., simulation, synthesis). We discuss how basic characteristics of GASPARD models can be verified to ensure the correctness of the specifications. We address the analysis of single assignment, functional determinism, and causality on the resulting synchronous models. We also illustrate synchronizability analysis on a simple application example for correctness issues. This study is the first attempt to relate explicitly multidimensional data structures to the synchronous paradigm.

A prototype engine, based on MDE has been developed to enable the automatic transformation of GASPARD descriptions into synchronous models [38]. A few sample demonstrations (item "Gaspard2 to Lustre") can be found at the address http://www2.lifl.fr/west/DaRTShortPresentations. They illustrate the design, the

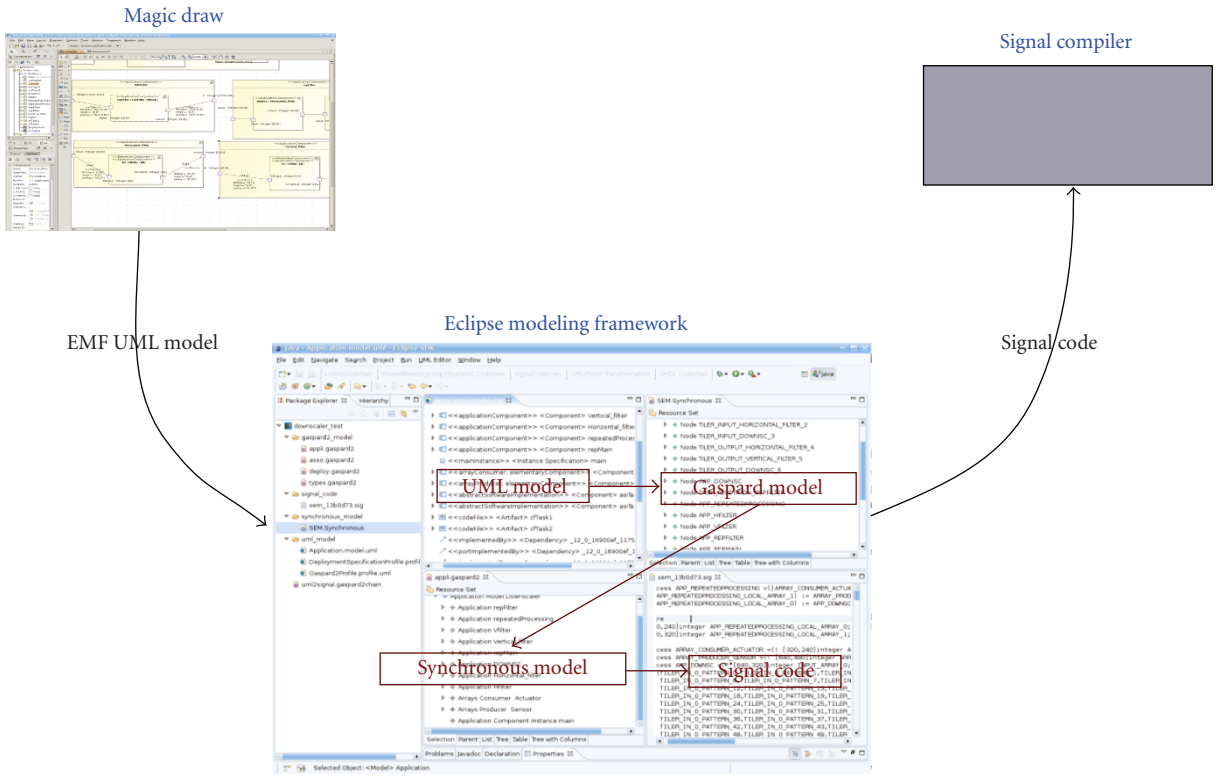


Figure 20: The architecture of the implementation tools.

transformation, and example of Lustre code generated from Gaspard models. The generated code is used to illustrate a functional simulation with the SIMEC simulator available in the Lustre environment, and causality analysis.

While this study shows very promising results, a main limitation is that the resulting synchronous models can be huge due to the explicit instantiation of Array-OL data-parallel constructs. This can reduce the applicability of more advanced analyses.

The perspectives of this work are manifold. First, we want to enhance the current control extension of GASPARD. Control has been introduced in the form of computation mode automata (inspired by synchronous dataflow languages) that define several ways of achieving different computations. The modes can differ either by the way data are accessed by tilers, or by the nature of the algorithm that applies on received data. Following the preliminary work of [35], we plan to make possible the specification of hierarchical and parallel automata defining modes and transitions between them. The resulting transition systems could be analyzed using techniques such as model-checking based on our transformation toward synchronous languages. A more constructive perspective is the use of discrete controller synthesis techniques. By introducing properly task control structures [46, 47], this technique can be used to generate automatically part of the task handler to enforce safety properties in the system under design. Another challenging question concerning the link between the fusion transformation of Array-OL [6] and synchronous clock calculi. The fusion transformation enables to deduce from any application description, a specification in which there is a unique top-level task in the hierarchy. It seems that the execution of a repetitive task model resulting from the fusion transformation matches the multidimensional time model proposed by Feautrier [48]. How could such a multidimensional time model be defined in synchronous languages? If it is possible, how could the associated clock calculus be defined? The answer to these questions will help us to identify a suitable clock notion for GASPARD and to benefit from the know-how of synchronous clock calculi.

REFERENCES

[1] A. Demeure and Y. Del Gallo, "An array approach for signal processing design," in *Proceedings of the Sophia-Antipolis Conference on Micro-Electronics (SAME '98)*, Sophia Antipolis, France, October 1998.

[2] The DaRT Team, http://gforge.inria.fr/projects/gaspard2.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[4] P. Boulet, "Array-OL revisited, multidimensional intensive signal processing specification," Tech. Rep. RR-6113, INRIA, Paris, France, February 2007.

[5] P. Boulet, "Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing," Tech. Rep. PR-6467, INRIA, Paris, France, March 2008.

[6] A. Amar, P. Boulet, and P. Dumont, "Projection of the Array-OL specification language onto the Kahn process network computation model," in *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*

(ISPAN '05), pp. 496–501, Las Vegas, Nev, USA, December 2005.

[7] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "POLYCHRONY for system design," *Journal of Circuits, Systems and Computers*, vol. 12, no. 3, pp. 261–303, 2003.

[8] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, vol. 19 of *Lecture Notes in Computer Science*, pp. 362–376, Paris, France, 1974.

[9] G. Kahn, "The semantics of simple language for parallel programming," in *Proceedings of the IFIP Congress*, vol. 74 of *Information Processing*, pp. 471–475, Stockholm, Sweden, August 1974.

[10] W. Wadge and E. Ashcroft, *LUCID, the Dataflow Programming Language*, Academic Press, San Diego, Calif, USA, 1985.

[11] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "LUSTRE: a declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*, pp. 178–188, ACM Press, Munich, Germany, January 1987.

[12] P. Caspi and M. Pouzet, "Synchronous Kahn networks," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pp. 226–238, Philadelphia, Pa, USA, May 1996.

[13] L. Besnard, T. Gautier, and P. L. Guernic, "SIGNAL V4—INRIA version: Reference Manual," 2007, http://www.irisa.fr/espresso/Polychrony.

[14] High Performance Fortran Forum, "High performance fortran language specification," January 1997, http://hpff.rice.edu/versions/hpf2/index.htm.

[15] B. B. Fraguela, J. Guo, G. Bikshandi, et al., "The hierarchically tiled arrays programming approach," in *Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-Time Support for Scalable Systems (LCR '04)*, vol. 81, pp. 1–12, Houston, Tex, USA, 2004.

[16] G. Almási, L. De Rose, B. B. Fraguela, J. Moreira, and D. Padua, "Programming for locality and parallelism with hierarchically tiled arrays," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC '03)*, vol. 2958 of *Lecture Notes in Computer Science*, pp. 162–176, College Station, Tex, USA, October 2003.

[17] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language," in *Proceedings of the 9th International Workshop on High-Level Programming Models and Supportive Environments (HIPS '04)*, vol. 9, pp. 52–60, Santa Fe, NM, USA, April 2004.

[18] E. Allen, D. Chase, J. Hallett, et al., "The fortress language specification, version 1.0 beta," March 2007, http://research.sun.com/projects/plrg/fortress.pdf.

[19] P. Charles, C. Grothoff, V. Saraswat, et al., "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, pp. 519–538, ACM Press, San Diego, Calif, USA, October 2005.

[20] W. Thies, M. Karczmarek, M. Gordon, et al., "StreamIt: a compiler for streaming applications," MIT/LCS Technical Memo MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, Mass, USA, December 2001.

[21] E. Lee, C. Hylands, J. Janneck, et al., "Overview of the ptolemy project," Technical Memorandum UCB/ERL M01/11, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Calif, USA, 2001.

[22] C. Mauras, *Alpha: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*, Ph.D. thesis, Université de Rennes I, Rennes, France, December 1989.

[23] D. Wilde, "The ALPHA language," Tech. Rep. 827, IRISA/INRIA, Rennes, France, 1994.

[24] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow: describing signal processing algorithm for parallel computation," in *Proceedings of the 32nd IEEE Computer Society International Conference (COMPCON '87)*, pp. 310–315, San Francisco, Calif, USA, February 1987.

[25] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '95)*, vol. 5, pp. 3255–3258, Detroit, Mich, USA, May 1995.

[26] E. A. Lee, "Mulitdimensional streams rooted in dataflow," in *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (PACT '93)*, vol. A-23 of *IFIP Transactions*, pp. 295–306, Orlando, Fla, USA, January 1993.

[27] P. K. Murthy and E. A. Lee, "Multidimensional synchronous dataflow," *IEEE Transactions on Signal Processing*, vol. 50, no. 8, pp. 2064–2079, 2002.

[28] J. Keinert, C. Haubelt, and J. Teich, "Windowed synchronous data flow," Tech. Rep. Co-Design Report 02, Department of Computer Science 12, Hardware-Software-Co-Design, University of Erlangen-Nuremberg, Erlangen, Germany, 2005.

[29] D.-I. Ko and S. S. Bhattacharyya, "Modeling of block-based DSP systems," *The Journal of VLSI Signal Processing*, vol. 40, no. 3, pp. 289–299, 2005.

[30] N. Halbwachs and D. Pilaud, "Use of a real-time declarative language for systolic array design and simulation," in *Proceedings of the International Workshop on Systolic Arrays*, Oxford, UK, July 1986.

[31] F. Rocheteau and N. Halbwachs, "POLLUX, a LUSTRE-based hardware design environment," in *Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures II*, P. Quinton and Y. Robert, Eds., pp. 335–346, Chateau de Bonas, Gers, France, June 1991.

[32] L. Morel, "Array iterators in Lustre: from a language extension to its exploitation in validation," *EURASIP Journal of Embedded Systems*, vol. 2007, Article ID 59130, 16 pages, 2007.

[33] P. Feautrier, "Array expansion," in *Proceedings of the 2nd International Conference on Supercomputing*, pp. 429–441, St. Malo, France, June 1988.

[34] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[35] O. Labbani, J. Dekeyser, P. Boulet, and E. Rutten, "Introducing control in the gaspard2 data-parallel metamodel: synchronous approach," in *Proceedings of the International Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES '05)*, Montego Bay, Jamaica, October 2005.

[36] I. Smarandache, T. Gautier, and P. Le Guernic, "Validation of mixed SIGNAL-ALPHA real-time systems through affine calculus on clock synchronisation constraints," in *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, vol. 1709 of *Lecture Notes In Computer Science*, pp. 1364–1383, London, UK, September 1999.

[37] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet, "N-sychronous Kahn networks," in *Proceedings of the 33th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '06)*, pp. 180–193, Charleston, SC, USA, January 2006.

[38] H. Yu, A. Gamatié, E. Rutten, and J.-L. Dekeyser, "Model transformations from a data parallel formalism towards synchronous languages," in *Embedded Systems Specification and Design Languages*, V. Eugenio, Ed., vol. 10 of *Lecture Notes Electrical Engineering*, chapter 13, pp. 183–198, Springer, London, UK, 2008.

[39] L. Morel, "Efficient compilation of array iterators for Lustre," in *Proceedings of the 1st Workshop on Synchronous Languages, Applications, and Programming (SLAP '02)*, vol. 65 of *Electronic Notes in Theoretical Computer Science*, pp. 19–26, Grenoble, France, April 2002.

[40] N. Halbwachs, P. Raymond, and C. Ratel, "Generating efficient code from data-flow programs," in *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, pp. 207–218, Passau, Germany, August 1991.

[41] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991.

[42] Intel Corporation, "Intel Quick Capture Technology for the Intel PXA27x Processor Family," 2004, http://www.intel.com/design/pca/applicationsprocessors/whitepapers/300873.htm.

[43] I. Smarandache, *Transformations affines d'horloges: application au codesign de systèmes temps réel en utilisant les langages SIGNAL et ALPHA*, Ph.D. thesis, Université de Rennes 1, Rennes, France, October 1998.

[44] N. Halbwachs, Y.-E. Proy, and P. Roumanoff, "Verification of real-time systems using linear relation analysis," *Formal Methods in System Design*, vol. 11, no. 2, pp. 157–185, 1997.

[45] A. A. Kountouris and P. Le Guernic, "Profiling of SIGNAL programs and its application in the timing evaluation of design implementations," in *Proceedings of the IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*, vol. 6, pp. 1–9, HP Labs, Bristol, UK, February 1996.

[46] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten, "Using controller-synthesis techniques to build property-enforcing layers," in *Proceedings of the 12th European Symposium on Programming Languages and Systems (ESOP '03)*, vol. 2618 of *Lecture Notes in Computer Science*, pp. 174–188, Warsaw, Poland, April 2003.

[47] G. Delaval and É. Rutten, "A domain-specific language for multi-task systems, applying discrete controller synthesis," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 84192, 17 pages, 2007.

[48] P. Feautrier, "Some efficient solutions to the affine scheduling problem—part II: multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.