

Synchronous Modeling of Data Intensive Applications

Abdoulaye Gamatié — Éric Rutten — Huafeng Yu — Pierre Boulet — Jean-Luc Dekeyser

N° 5876

Avril 2006

Thème COM



*Rapport
de recherche*

Synchronous Modeling of Data Intensive Applications

Abdoulaye Gamatié^{*}, Éric Rutten[†], Huafeng Yu[‡], Pierre Boulet[§], Jean-Luc Dekeyser[¶]

Thème COM — Systèmes communicants
Projet DaRT

Rapport de recherche n° 5876 — Avril 2006 — 21 pages

Abstract: In this report, we present the first results of a study on the modeling of data-intensive parallel applications following the synchronous approach. More precisely, we consider the GASPARD extension of ARRAY-OL, which is dedicated to System-on-Chip codesign. We define an associated synchronous dataflow equational model that enables to address several design correctness issues (e.g. verification of frequency / latency constraints) using the formal tools and techniques provided by the synchronous technology. We particularly illustrate a synchronizability analysis using affine clock systems. Directions are drawn from these bases towards modeling hierarchical applications, and adding control automata involving verification.

Key-words: Intensive data processing, modeling, GASPARD, synchronous dataflow

* abdoulaye.gamatie@lifl.fr

† eric.rutten@lifl.fr

‡ huafeng.yu@lifl.fr

§ pierre.boulet@lifl.fr

¶ Jean-Luc.Dekeyser@lifl.fr

Modélisation synchrone d'applications de traitement de données intensives

Résumé : Dans ce rapport, nous présentons les premiers résultats d'une étude sur la modélisation d'applications parallèles de traitement de données intensives, basée sur l'approche synchrone. Plus exactement, nous considérons l'extension GASPARD d'ARRAY-OL, qui est dédiée à la conception conjointe de systèmes intégrés sur puce. Nous définissons un modèle flot de données synchrone équationnel associé, qui permet d'aborder plusieurs questions liées à la correction lors de la conception (par exemple, vérification de contraintes de latence ou de fréquence), en utilisant les outils et techniques formelles offerts par la technologie synchrone. Nous illustrons particulièrement une analyse de synchronisabilité en utilisant les systèmes d'horloges affines. Des perspectives sont ensuite mentionnées concernant la modélisation d'applications hiérarchiques, et l'ajout d'automates de contrôle impliquant la vérification.

Mots-clés : Traitement de données intensives, modélisation, GASPARD, flot de données synchrone

Contents

1	Introduction	4
2	Data-parallelism and synchronous models	4
2.1	Data-intensive applications and GASPARD	4
2.1.1	The global model	5
2.1.2	The local model	6
2.2	Synchronous models	7
2.2.1	Synchronous dataflow models	8
2.2.2	The oversampling mechanism	8
2.2.3	Affine clocks	9
2.3	Related works	11
3	Synchronous model of GASPARD	11
3.1	Parallel model	12
3.1.1	Modeling of data	12
3.1.2	Modeling of one repetition in the repetition space	12
3.1.3	Restructuring the simple parallel model	13
3.2	Serialized model	14
3.2.1	General view	14
3.2.2	Construction of pattern flows from arrays	15
3.2.3	Reconstruction of arrays from pattern flows	16
4	Dealing with validation issues	16
4.1	Synchronizability analysis using affine clocks	17
4.2	Other analysis	18
5	Discussions	19
6	Conclusions	19

1 Introduction

Computing and analyzing large amounts of data play an increasingly important role in embedded systems. The concerned applications often perform calculations 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, etc. Highly desirable design approaches for such applications are those providing users with well-adapted concepts to represent the data manipulations, and techniques that trustworthily guarantee important implementation requirements.

A major goal of the GASPARD framework is to fill this demand. Its associated specification formalism, called ARRAY-OL [7], has been originally proposed within an industrial context by Thomson Marconi Sonar, for the description of data-intensive applications manipulating regular multidimensional data structures. However, ARRAY-OL does not allow to deal with non functional issues, for instance, temporal constraints imposed by the environment on applications, or control of computations according to different modes or configurations.

Motivations. We propose a synchronous model of data intensive applications based on the GASPARD extension of ARRAY-OL, which manipulates data through multidimensional, toroidal and possibly infinite arrays. The specifications provided by this formalism express data dependencies only based on values (i.e. true data dependencies), thus yield a minimal execution order. They adopt a single assignment style. Moreover, they are independent from architecture details. All these features confer to ARRAY-OL a powerful expressiveness for data manipulation. The model we propose aims at providing on the one hand the same expressiveness as GASPARD descriptions, and on the other hand the possibility to deal with control and non functional constraints in data-intensive applications. It therefore allows us to explore at a higher level different refinements of initial GASPARD descriptions w.r.t. the constraints from application environments or target implementation platforms. This exploration is supported by the synchronous technology, which suits for the trusted design due to its mathematical foundations. In particular, we address synchronizability issues in case of GASPARD models using affine clock systems defined in the synchronous language SIGNAL.

Outline. In the sequel, Section 2 presents the general background of our study. It first provides an overview of GASPARD and ARRAY-OL, then introduces the basic synchronous concepts for our models. Moreover, it gives some related works. Section 3 focuses on the synchronous modeling of GASPARD applications. The usefulness of such a modeling is carried out in Section 4. We show how existing results of the synchronous approach can help to deal with critical design issues for GASPARD applications. In Section 5, we discuss our solution and we detail some of the current challenges. Finally, concluding remarks are given in Section 6.

2 Data-parallelism and synchronous models

2.1 Data-intensive applications and GASPARD

GASPARD (Graphical Array Specification for Parallel and Distributed Computing) [21] is an environment that implements a codesign methodology for System-on-Chip, based on the *Model-Driven Engineering* approach as illustrated by Figure 1. It proposes a UML profile (integrated

in the current OMG MARTE profile for real-time and embedded systems) allowing designers to model both data-intensive applications and their architectures. An association mechanism is provided for the two aspects, together with a set of transformations for simulation and synthesis. Our modeling approach aims at taking into account all these features of GASPARD so as to be able to prove the correctness of the models manipulated during its design methodology.

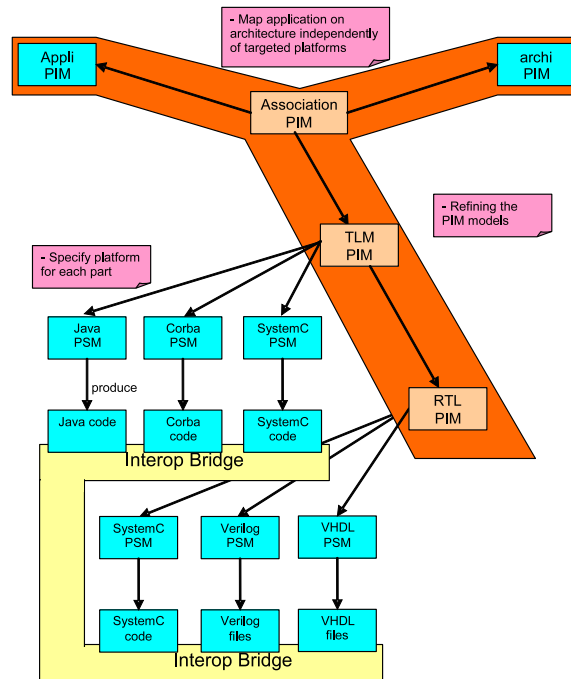


Figure 1: SoC design based on GASPARD.

The underlying specification formalism of GASPARD, called ARRAY-OL (Array Oriented Language) [7] [20] allows one to model intensive signal processing applications manipulating large amounts of data in a regular way. It adopts multidimensional representations that enable to express the whole potential parallelism available in target applications. Data are structured into arrays (that may be possibly infinite). A task consumes and produces arrays by “pieces” of the same size called *patterns*. The different tasks are connected to each other through *data dependencies*. When a dependency is specified between two tasks, it means that one of them requires data to be produced by the other before executing. These dependencies initially yield a minimal partial execution order. Applications can be hierarchically composed at different specification levels. In practice, their specification consists of a *global model* and a *local model*, presented in the next sections.

2.1.1 The global model

The global model consists of a directed acyclic graph where nodes represent tasks and edges represent multidimensional arrays. There is no restriction on the number of incoming or out-

going arrays. These arrays are assumed to be *toroidal*, i.e., their elements can be consumed or produced modulo the array size. While the global model provides information to schedule the tasks, it does not express the data parallelism present in these tasks. This is described by the local model.

2.1.2 The local model

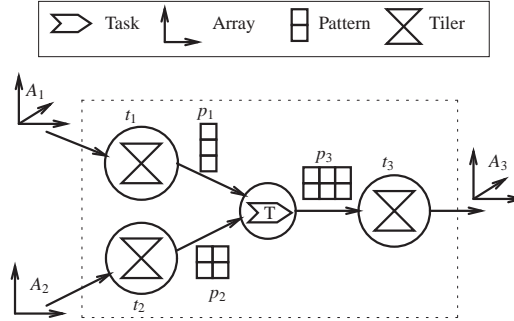


Figure 2: A local model in ARRAY-OL.

The local model describes the data parallelism expressed by *repetitions* (see Figure 2). A task is defined by a repetition constructor, where task repetitions (or *instances*) are independent from each other. Every instance is applied to a subset of elements (i.e. patterns) from input arrays to produce elements stored in output arrays. The way a task consumes and produces arrays can be analyzed through a couple (*task, array*). Such couples are referred to as *half-tasks*. Let (T, A_i) be a half-task. If A_i is an input array, T takes patterns from A_i to achieve its processing, otherwise T stores its calculated patterns in A_i . The size and shape of patterns associated with an array are the same from one repetition to another. Patterns can be themselves multidimensional arrays. Their construction is achieved via *tilers*, which contain the following information: \vec{o} - the origin of the reference pattern, \vec{d} - the shape of the pattern (size of all its dimensions), P - the paving matrix (how patterns cover arrays), F - the fitting matrix (how array elements fill patterns), and \vec{m} - the shape of arrays (size of all its dimensions).

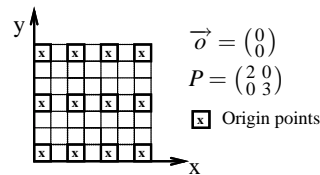


Figure 3: An array paving.

Each couple (*task, array*) is therefore associated with a *tiler*. To enumerate the different patterns, each half-task has, via its tiler, a *paving matrix* P and a starting point represented by the *origin* \vec{o} . The paving matrix enables to identify the origin of every pattern, associated with each task instance. Let us consider a two-dimensional array with $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ as origin point and $\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$

as paving matrix. The pattern constructions start from positions noted “x” in figure 3.

Equation 1 states that the coordinates of each first point of a pattern are calculated as the sum of the coordinates of the origin point and a linear combination of paving vectors, the whole modulo the size of the array (since arrays are toroidal).

$$\forall \vec{x}_q, \vec{0} \leq \vec{x}_q < \vec{Q}, \vec{r}_q = (\vec{o} + P \times \vec{x}_q) \mod \vec{m} \quad (1)$$

\vec{x}_q denotes a pattern with index q . We refer to as *repetition space*, the set of all possible indices q . \vec{Q} contains the bounds of repetition for each vector of the paving matrix (i.e. it delimits the repetition space). Finally, \vec{r}_q represents the origin point of the pattern \vec{x}_q .

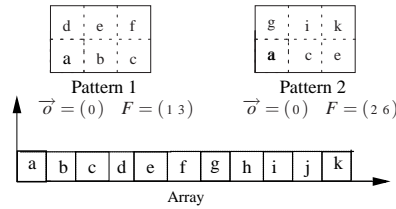


Figure 4: Fitting examples.

The *fitting matrix* allows one to determine array elements associated with each pattern. Figure 4 illustrates two simple examples using an monodimensional array with (0) as origin, where the associated pattern is a two-dimensional array of shape $\vec{d} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$. A pattern can have more dimensions than a consumed/produced array. In the first case the fitting matrix is $(1\ 3)$. Each vector of this matrix is used to fill one dimension of the pattern. In the second case, the fitting matrix is $(2\ 6)$. We observe that the elements associated with a pattern are not necessarily consecutive elements in an array. These elements are determined via the sum of the coordinates of the first element of the pattern and a linear combination of the fitting matrix, the whole modulo the size of the array (see Equation. 2).

$$\forall \vec{x}_d, \vec{0} \leq \vec{x}_d < \vec{d}, (\vec{r}_d + F \times \vec{x}_d) \mod \vec{m} \quad (2)$$

Here, \vec{x}_d denotes an element with index d in a pattern.

According to the specification of the local model, patterns can be themselves multidimensional arrays. This enables *hierarchical* definitions of applications. For instance, the single task T represented in Figure 2 can be composed of sub-tasks whose incoming and outgoing arrays are the patterns p_i of this task. In the next, we invariably use ARRAY-OL and GASPARD to designate the same models.

2.2 Synchronous models

The *synchronous approach* [2] has been proposed in order to provide formal concepts that favor the trusted design of embedded real-time systems. Its basic assumption is that computation and communication are instantaneous (referred to as “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 (i.e., duration has a significant role).

2.2.1 Synchronous dataflow models

Historically, the origin of dataflow languages can be associated with earlier studies on dataflow models started in 70's [8] [11] [23]. This is the case for declarative synchronous languages such as LUSTRE [4], LUCID SYNCHRONE [5] or SIGNAL [14]. While LUSTRE and LUCID SYNCHRONE adopt a functional style, SIGNAL is relational. In these languages, manipulated data consist of unbounded sequences of values. A few operators are provided to define how these sequences are related to each other. This is basically expressed using equations. In the remainder of this paper, although the scope of our study aims at synchronous dataflow models in general, we mainly consider the SIGNAL language for illustration.

An introduction to SIGNAL. The language handles unbounded series of typed values $(x_t)_{t \in \mathbb{N}}$, 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 noted \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*, noted $\hat{x} = \hat{y}$. A *process* is a system of equations over signals that specifies relations between values and clocks of the involved signals. A *program* is a process. SIGNAL relies on a six primitive constructs used in equations as follows:

- *Relations*: $y := f(x_1, \dots, x_n) \stackrel{def}{=} y_t \neq \perp \Leftrightarrow x_{1t} \neq \perp \Leftrightarrow \dots \Leftrightarrow x_{nt} \neq \perp$, and $\forall t: y_t = f(x_{1t}, \dots, x_{nt})$.
- *Delay*: $y := x \ \$ \ 1 \ \text{init } c \stackrel{def}{=} x_t \neq \perp \Leftrightarrow y_t \neq \perp, \forall t > 0: y_t = x_{t-1}, y_0 = c$.
- *Undersampling*: $y := x \ \text{when } b \stackrel{def}{=} y_t = x_t \text{ if } b_t = \text{true}, \text{ else } y_t = \perp$. The expression $y := \text{when } b$ is equivalent to $y := b \ \text{when } b$.
- *Deterministic merging*: $z := x \ \text{default } y \stackrel{def}{=} z_t = x_t \text{ if } x_t \neq \perp, \text{ else } z_t = y_t$.
- *Composition*: $(P \ | \ Q) \stackrel{def}{=} \text{union of equations of } P \ \text{and } Q$.
- *Hiding*: $P \ \text{where } x \stackrel{def}{=} x$ is local to the process P .

These constructs are enough expressive to derive other constructs for comfort and structuring. SIGNAL also offers a process frame, which enables the definition of sub-processes (declared in the *where* sub-part, see Figures 9 and 10 for example). Sub-processes that are only specified by an interface without internal behavior are considered as external, and may be separately compiled processes or physical components.

2.2.2 The oversampling mechanism

A useful notion that will be considered to model ARRAY-OL specifications is *oversampling* mechanism. It consists of a temporal refinement of a given clock c_1 , which yields another clock c_2 , faster than c_1 (i.e. c_2 contains more instants than c_1). To define this mechanism, let us

consider the following sets: $\mathcal{B} = \{ff, tt\}$ (ff and tt respectively denote *false* and *true*); \mathcal{V} a value domain (where $\mathcal{B} \subseteq \mathcal{V}$); and \mathbb{T} a dense set associated with a partial order relation \leq . The elements of \mathbb{T} are called *tags*. Each pair of tags $(t_1, t_2) \in \mathbb{T}^2$ admits a lower bound. A *set of observation points* \mathcal{T} is a set of tags s.t.: i) $\mathcal{T} \subset \mathbb{T}$, ii) \mathcal{T} is countable, iii) each pair of tags admits a lower bound in \mathcal{T} . Finally, a *chain* $C \subset \mathcal{T}$ is a totally ordered set, which admits a lower bound.

Definition 1 (signal, clock, oversampling)

- A *signal* s is a partial function $s \in \mathcal{T} \rightarrow \mathcal{V}$, which associates values with observation points of a chain. A *clock* c is a special signal s.t. $c \in \mathcal{T} \rightarrow \{tt\}$.
- Let c_1 denote a clock, the k -oversampling of c_1 derives a clock c_2 from c_1 , noted $c_2 = k \uparrow c_1$, s.t. c_2 contains k instants per instant in c_1 , where k is an integer.

```

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

      c1:  tt  ⊥  ⊥  ⊥  tt  ⊥  ⊥  ⊥  ...
count:  3  2  1  0  3  2  1  0  ...
pre_count:  0  3  2  1  0  3  2  1  ...
      c2:  tt  tt  tt  tt  tt  tt  tt  tt  ...

```

Figure 5: Clock oversampling: $c_2 = 4 \uparrow c_1$.

In the SIGNAL program given in Figure 5, called `k_overspl`, k is a constant integer parameter (line 1). Signals c_1 and c_2 respectively denote input and output clocks (line 2), where c_2 is a k -oversampling of c_1 . The `event` type is associated with clocks. It is equivalent to a boolean type where the only taken value is *true*. The local signals `count` and `pre_count` serve as counter to define k instants in c_2 per instant in c_1 (lines 3 and 4).

Note that the oversampling mechanism can be specified in LUCID SYNCHRONE, but not in LUSTRE.

2.2.3 Affine clocks

We introduce the *affine clock* notion, which allows one to deal with synchronizability issues in a more relaxed way than usually in synchronous languages [18].

Definition 2 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 d^{th} instant, starting with the ϕ^{th} (see fig. 6).

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

Figure 6: c_1 and c_2 are in $(3, 1, 4)$ -affine relation.

Clocks c_1 and c_2 are said to be in (n, ϕ, d) -affine relation, noted as $c_1 \xrightarrow{(n, \phi, d)} c_2$.

Affine relations provide a relaxed vision of the usual synchronous model in which different signals are synchronous if and only if they have identical clocks. In affine clock systems, two different signals are said to be synchronizable if there is a dataflow preserving way to make them actually synchronous.

Affine transformations on SIGNAL variables are mainly described using three derived operators:

- *Affine undersampling*: $y := \text{affine_sample}\{\phi, d\}(x)$, where signals x and y are of the same type, $\phi \in \mathbb{Z}^+$ and $d \in \mathbb{Z}^+ - \{0\}$. The signal y is defined as an undersampling with phase ϕ and period d on the signal x . In the following trace, $\phi = 2$ and $d = 3$:

x:	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	...
y:	⊥	⊥	x_2	⊥	⊥	x_5	⊥	⊥	x_8	⊥	...

- *Affine relation*: $\text{affine_clock_relation}\{n, \phi, d\}(x, y)$, where signals x and y are of any type, $\phi \in \mathbb{Z}$ and $n, d > 0$. It defines a (n, ϕ, d) -affine relation between the clocks of x and y .
- *Affine oversampling*: $y := \text{affine_unsample}\{n, \phi\}(x, z)$, where signals x , y and z are of the same type, $n > 0$ and $\phi \in \mathbb{Z}$. It defines y as an affine oversampling using x and z as illustrated by the following trace with $n = 3$ and $\phi = 2$:

x:	x_0	⊥	x_1	⊥	⊥	x_2	⊥	⊥	x_3	⊥	...
z:	⊥	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	...
y:	⊥	z_0	x_1	z_2	z_3	x_2	z_5	z_6	x_3	z_8	...

The `affine_clock_relation` and `affine_unsample` operators can be straightforwardly expressed using the `affine_sampling` operator [18]. The SIGNAL *clock calculus* (i.e. its static analysis phase) has been extended in order to address such synchronizability issues with the associated compiler. Affine clock relations are only available in SIGNAL. However, a notion of periodic clock has been recently proposed in LUCID SYNCHRONE [6]. It defines another vision of synchronizability in synchronous dataflow models.

2.3 Related works

In [19], Smarandache *et al.* address the validation of real-time systems by considering the functional data parallel language ALPHA [15] and SIGNAL. In their approach, intensive numerical computations are expressed in ALPHA while the control (the clock constraints resulting from ALPHA descriptions after transformations) is conveyed to SIGNAL. The regularity of 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. In Section 4, we use the same criteria to analyze a given GASPARD model. Similar concepts can be found in [6] where a synchronous model is defined in order to address the correct-by-construction development of high performance stream-processing applications. It particularly relies on a domain specific knowledge consisting of *periodic evolution of streams*. This model allows to automatically synthesize communications between processes with periodic clocks that are not strictly synchronous.

Other languages such as OTTO E MEZZO [16] and STREAMIT [22] can be also mentioned. The former allows to describe behaviors of dynamical systems. It uses clock information in the code generation (e.g. in C or towards a SIMD abstract machine). The second language proposes an efficient compilation technique of streaming applications and mapping to different kinds of architectures (monoprocessor, grid-based processors, etc.). Both languages share features with synchronous dataflow languages. Similarly to our approach, they aim at data-intensive applications.

Previous studies on the transformation of ARRAY-OL descriptions [20] [9] [1] address compilation issues for efficient code generation, and the relation between ARRAY-OL and other close models (e.g. Multidimensional Synchronous Dataflow Model of Lee *et al.* and Kahn process networks). Finally, the earlier work of Labbani *et al.* on the introduction of control in GASPARD [13] can be noted. In this work, the authors discuss the definition of instants at which mode changes occur in data-intensive applications. The synchronous modeling of their results will yield descriptions with conditioned computations and transition systems, where synchronous verification and compilation techniques can be extensively used.

3 Synchronous model of GASPARD

We propose a modeling of GASPARD applications using synchronous dataflow languages in order to address correctness issues (synchronizability, constraints on latencies, etc.).

Before exposing our approach, we make a few important observations. First, the models we propose aim at taking into account refinement constraints on GASPARD models. These constraints mainly include implementation properties: platform and environment constraints. In the first case, the fully parallel execution model adopted from the ARRAY-OL viewpoint is broken at the implementation level since the number of available processors in a platform does not often enable to achieve such parallel execution. As a result, serialized execution models become necessary. Second, the input/output data manipulated by applications via infinite arrays are produced/consumed by devices such as sensors and actuators. The production and consumption rates characterizing these devices represent the major part of what we refer to as environment constraints. It therefore appears natural to represent those infinite arrays by a flow

data (or arrays), which induces some logical time scale. The resulting model becomes richer than a GASPARD model for analysis.

On the other hand, to define our models, we only consider applications that have been initially transformed using the so-called *fusion* algorithm in ARRAY-OL [1]. Roughly speaking, this algorithm aims at reducing a set of tasks into a single task. The resulting task is hierarchical and calls the initial tasks as subtasks (see Figure 11 for an illustration). Furthermore, it enables to transform an incoming multidimensional infinite array at the top level of the hierarchy into a flow of incoming sub-arrays. This algorithm has been successfully used to define projections of ARRAY-OL specifications on Kahn process networks and PTOLEMY models.

In the sequel, we first show a synchronous model that fully preserves the parallelism of an ARRAY-OL specification. This model can be used for formal verification using classical synchronous techniques (e.g. compilation, model-checking). Then, we present a refined version of such a model, where executions are sequential. This second vision can be typically associated with a mono-processor execution. We briefly discuss the combination of both models. Environment constraints will be addressed in Section 4. Finally, for the sake of clarity, we restrict ourselves to the top level in the hierarchy of an application after having applied the fusion algorithm. If we consider the ARRAY-OL task T depicted by Figure 2 as such a top level task, two possible models will be distinguished: a *parallel model*, which specifies exactly the same data-parallelism as T (see Section 3.1), and a *serialized model*, which sequentializes the execution of T (see Section 3.2).

3.1 Parallel model

The synchronous dataflow models are quite close in structure to the GASPARD models. Hence, the modeling of the latter in the former, using equations and synchronous composition, is quite simple.

3.1.1 Modeling of data

ARRAY-OL arrays are modeled into flows of array type. A particularity of arrays in ARRAY-OL is that they can have (at most) one infinite dimension, and that there is no explicit representation of time whatsoever. It occurs that the infinite dimension of an array is the usual way to represent an infinite flow of arrays of the remaining dimensions. Therefore, we will enforce this representation concretely by going to array flows.

3.1.2 Modeling of one repetition in the repetition space

For a task transforming input data arrays A_1 and A_2 into an output data array A_3 , this can be done by an equation of the following form:

$$A_3[\langle ind_3^j \rangle] := T(A_1[\langle ind_1^j \rangle], A_2[\langle ind_2^j \rangle]) \quad (3)$$

where ind_i^j denotes indexes corresponding to a point j in the repetition space; and T is the synchronous model of the computation part, which can be seen as a call to an external function. We also suppose that synchronous array flows can be assigned by their elements. For instance, in SIGNAL, this can be expressed using the derived construct `next` [3].

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

Another particularity is that ARRAY-OL represents only data dependencies, hence leaving all the potential parallelism in the specification. 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. We are conforming to this property of ARRAY-OL simply by using synchronous composition between models of all the repetitions, thereby inducing no order whatsoever between them.

One can observe that it is possible to have a more compact expression of this simple model, according to the array-specific operators available in the equational synchronous language considered. A *map* of the function T on the array enables a more compact notation [17]. It can be noted that the model we present here is meant to be intuitive and simple, and is certainly optimizable. However, the optimizations of the synchronous equations can be quite different according to the intended target operations, e.g. code generation or model-checking.

3.1.3 Restructuring the simple parallel model

This model can be restructured, in order to better map the intrinsic structure of ARRAY-OL repetitive task, with input tilers, computations, and output tilers. One advantage is to have a structural transformation from ARRAY-OL to synchronous equations, which is implementable automatically as a simple translator. Another advantage is related to the introduction of control, which is a direct perspective of this work, following the preliminary results in [13]. Indeed, if one considers controlling such a task with automata, where the states correspond to a configuration or a mode, and transitions switch between these modes which all have the same input-output interface, then it is possible to have a control for each of the tilers or computation components of a task.

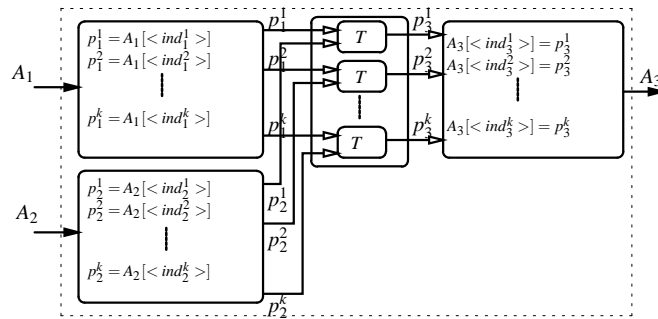


Figure 7: Modeling of tilers and parallel tasks.

This restructuring relies on the fact that it is costless to introduce intermediary signals, as they can be compiled away by the compilers and optimizers, and also on the properties of synchronous composition, which is associative and commutative. The above equation 3 can be separated in three parts, corresponding respectively to:

- the input tilers, composing together the equations performing the extraction of the input patterns for each point in the repetition space: $p_1^j := A_1[\langle ind_1^j \rangle]$ and $p_2^j := A_2[\langle ind_2^j \rangle]$.
- the computation, producing the output patterns for each point in the repetition space: $p_3^j := T(p_1^j, p_2^j)$.
- the output tilers, composing together the equations performing the insertion of the output patterns for each point in the repetition space: $A_3[\langle ind_3^j \rangle] := p_3^j$.

This way, we obtain a model as illustrated in Figure 7.

3.2 Serialized model

3.2.1 General view

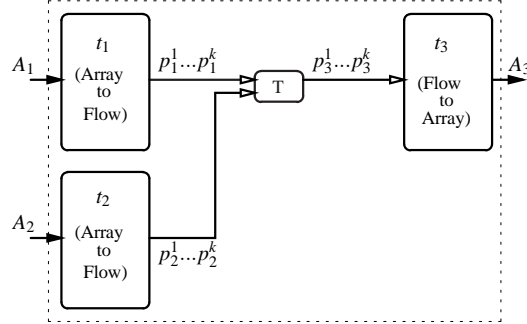


Figure 8: A serialized model.

While the parallel model fully preserves the data-parallelism present in ARRAY-OL specifications, the model introduced in this section provides a refined view of such specifications. As shown by Figure 8, we distinguish three basic sub-parts: a single task instance T (which can be still seen as an external function) and two kinds of components referred to as *Array to Flow* and *Flow to Array*. T receives its input patterns via a pattern flow from *Array to Flow* and sends its output patterns to *Flow to Array*, also via a pattern flow of the same length as the one given by *Array to Flow*.

In the next, we concentrate on the definition of *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 $\{t_{i,i \in \mathbb{N}}\}$ of a given clock c_1 . Then, for each t_i , the pattern production algorithm is applied to the received arrays: the task instance T is provided with the pair or patterns (p_1^i, p_2^i) and instantaneously produces the pattern p_3^i . The resulting pattern flow is therefore associated with a clock $c_2 = k \uparrow c_1$. The constant integer k corresponds to the number of paving iterations specified in tilers. It is directly derived from ARRAY-OL specifications.

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

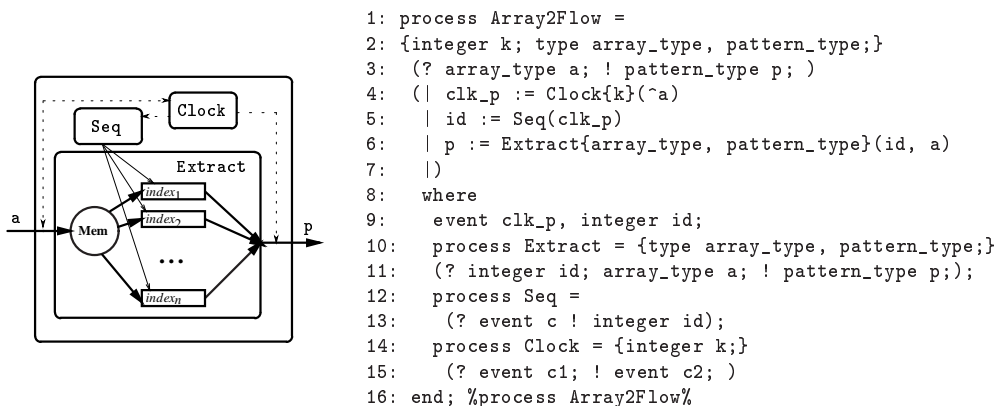


Figure 9: Construction of pattern flows from arrays.

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 patterns. Similarly, in *Flow to Array*, the incoming patterns must be accumulated locally until the k expected patterns become available. Then the output array can be produced.
2. *consumption and production rates*: in both components, we describe the frequencies at which patterns 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 pattern 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 patterns to be scheduled every instant t_i of the clock associated with a pattern flow.

3.2.2 Construction of pattern flows from arrays

The *Array to Flow* component is described by Figure 9. The illustration given on the left-hand side is encoded by the SIGNAL program shown on the right-hand side.

Three parts are distinguished. First, the `Clock` sub-process (line 4 in the program) defines the pattern 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` sub-process (line 6) is used to memorize an incoming array from which it extracts the patterns p . Finally, the `Seq` sub-process (line 5) describes in which order patterns are gathered from the input array. It produces pattern identifiers id at the same rate as the event `clk_p` defined by `Clock`. These identifiers are used by the `Extract` sub-process to produce patterns. The pattern enumeration strategy adopted in `Seq` can be decided from several viewpoints:

- *Environment*: the presence of sensors/actuators imposes particular orders following which data are received/produced.
- *Application*: in some algorithms, each computation step requires the results obtained at previous steps, hence leading to precedence constraints between task instances. Operating modes in some applications are also source of flows [13].

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

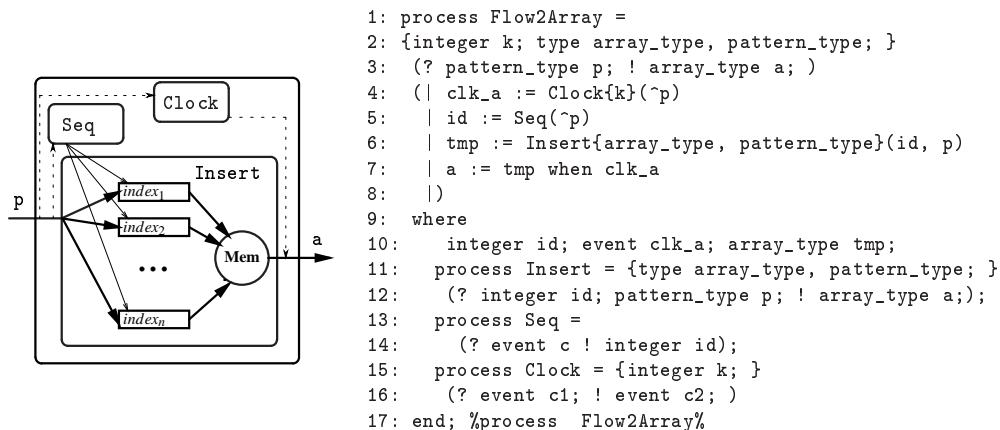


Figure 10: Construction of arrays from a pattern flows.

3.2.3 Reconstruction of arrays from pattern flows

The description of the *Flow to Array* component (see Figure 10) is obtained in a similar way as *Array to Flow*. We distinguish similar parts as *Array to Flow*.

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

From the above description, we observe that the parallel and serialized models are correct-by-construction. They offer the bases to represent GASPARD applications from high-level views (i.e. with a maximal parallelism) to more refined views (i.e. with sequential execution). More generally, one can imagine mixed models that combine the two models. These models can therefore be considered for further refinements, which take into account environment constraints. This issue is addressed in the next section.

4 Dealing with validation issues

We present how the synchronous models obtained from the previous section are used to address design correctness issues for GASPARD models.

4.1 Synchronizability analysis using affine clocks

We experiment affine clock systems, introduced in Section 2.2.3, on a simple application example in order to analyze constraints on data consumption and production rates.

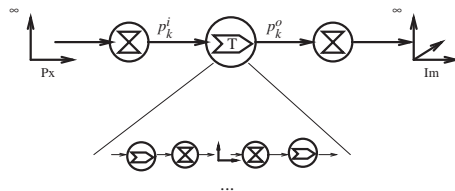


Figure 11: A hierarchical application model.

In Figure 11, we have illustrated a GASPARD application, which is dedicated to image processing. It takes pixels from an infinite array of pixels (denoted by Px), then performs a treatment on these pixels, and finally produces transformed pixels, which are combined to define an infinite array of images (denoted by Im). The application consists of a hierarchical GASPARD model obtained after the *fusion* transformation [1]. The resulting model allows us to consider flows of input and output arrays in the application. Here, the input p_k^i and output p_k^o respectively denote blocks of pixels (represented by arrays). Calculations are performed on such arrays in the lower layers of the hierarchy. This GASPARD description can be straightforwardly modeled by using exclusively or by combining the serialized and parallel synchronous models. So, we rather focus on the analysis of the resulting model by taking into account environment constraints.

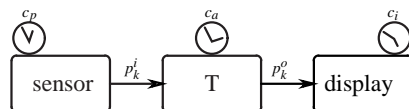


Figure 12: Application with its environment.

Let us consider the application from a refined point of view, where its environment is also modeled. We have to precise how input pixels and output images are produced. This is typically what GASPARD cannot currently enable to describe because of its assumption on the availability of the whole infinite arrays Px and Im (see Figure 11). For a more realistic model of the application, we have to take into account external constraints on pixel and image production rates. For instance, Px and Im can be respectively seen as the outputs of a *sensor* and a *display*, which may have *a priori* different clock rates (see Figure 12). Our initial description of the application can be consequently enhanced by composing its associated synchronous model with models of sensor and display. The clocks c_p , c_a and c_i respectively provide the pixel production rate in the sensor, the bloc computation frequency within the application, and the image production rate in the display. The resulting model becomes enough relevant to be considered for analysis. From this model, we can derive the following constraints:

- C_1 : c_a is an affine undersampling of c_p since every input of the global task T consists of a block of pixels with the same size, $c_p \xrightarrow{(1, \phi_1, d_1)} c_a$;
- C_2 : c_i is an affine undersampling of c_a since images are produced periodically by the display, from the same number of blocks of pixels (emitted by T), $c_a \xrightarrow{(1, \phi_2, d_2)} c_i$;

Now, let us consider a given external constraint, C_3 , which imposes a particular image production rate, denoted by a new clock c'_i , from c_p such that: $c_p \xrightarrow{(1, \phi_3, d_3)} c'_i$.

The synchronizability between clocks c'_i and c_i remains to be established w.r.t. the above constraints. It can be addressed using affine clock systems. From C_1 , C_2 and C_3 , this synchronizability is checked by using the following property (proved in [18], and now implemented in the SIGNAL compiler):

$$c'_i \text{ and } c_i \text{ are synchronizable} \Leftrightarrow \begin{cases} \phi_1 + d_1 \phi_2 = \phi_3 \\ d_1 d_2 = d_3 \end{cases} \quad (4)$$

The steps of the clock c_a correspond to the paving iterations of the global task T . So, given some fixed values of the frequency of clocks c_p and c_i (imposed by the application environment) verifying constraint C_3 , the way c_a is defined must allow to satisfy Equation 4 in order to ensure the synchronizability between c'_i and c_i . This issue can be solved quite easily with synchronous models while it is not possible with GASPARD only. The result of this analysis can be therefore used to adjust paving iteration properties so as to satisfy environment constraints.

Note that in addition to the above synchronizability analysis, there is similar analysis proposed in LUCID SYNCHRONE [6]. However, a notion of periodic clocks is considered instead of affine clocks. Synchronous GASPARD models defined in LUCID SYNCHRONE make it possible to access this facility.

4.2 Other analysis

Among the other available 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 [12]. 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 co-simulation of both programs therefore provides 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 within its design flow.

Finally, we 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.

All these features of the synchronous technology combined with GASPARD facilities promote a trustworthy design activity for data-intensive applications.

5 Discussions

The approach presented in this paper exposes our first results on modeling and validation of data-intensive applications using concepts of the synchronous approach. These results are very promising. They provide an interesting basis to relate two kinds of specification models: the ARRAY-OL data-parallel language and synchronous dataflow languages. These models offer altogether powerful concepts to cope with massively parallel applications. ARRAY-OL is very expressive and allows to define system architecture within GASPARD, while synchronous languages favor formal validation for the trusted design.

In the current status of our approach, the proposed models are applied after a specific transformation of ARRAY-OL specifications, called fusion. This permits to build correct-by-construction models. However, for generality, it will be interesting to explore how to define an equivalent model without assuming such a transformation. This leads to a very challenging question concerning the link between the fusion and synchronous clock calculi. It seems that the repetition of the task hierarchy resulting from the fusion transformation in ARRAY-OL matches the multidimensional time model proposed by Feautrier [10]. To the best of our knowledge, such a time model is not currently available in any synchronous language.

The second challenging issue concerns the introduction of mode automata in our models. Existing work [13] identified the basic concepts to be taken into account. In particular, the granularity of the control, and the fine grain control of tilers and computations, can produce quite complex models. This is where we will have clocks with a variety of conditioned relations, and transition systems that fully take advantage of the synchronous analysis, compilation, and verification techniques.

6 Conclusions

In this study, we propose a synchronous model for data-intensive applications within the GASPARD environment, which is dedicated to system-on-chip codesign. The GASPARD underlying specification language, ARRAY-OL, adopts a particular style where infinite multidimensional arrays are manipulated. We show how models described using such a language can be modeled using synchronous dataflow languages. 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 illustrate a 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.

Several issues remain to be explored in future works amongst which:

- investigating the relationship between, on the one hand, the ARRAY-OL expressiveness concerning regular massive parallelism, and on the other hand synchronous clock calculi, especially when they are extended with affine or periodic clocks;
- introducing control in GASPARD applications, at different levels of granularity, in order to exploit synchronous clock calculi and model-checking or discrete controller synthesis.

References

- [1] 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 International Symposium on Parallel Architectures, Algorithms, and Networks, Las Vegas, Nevada, USA*, December 2005.
- [2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.
- [3] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL reference manual. www.irisa.fr/espresso/Polychrony.
- [4] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. 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)*, pages 178–188. ACM Press, 1987.
- [5] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming*, pages 226–238, 1996.
- [6] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous Kahn networks. In *ACM Symp. on Principles of Programming Languages (PoPL'06)*, Charleston, South Carolina, USA, January 2006.
- [7] A. Demeure and Y. Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis conference on Micro-Electronics (SAME'98), System-on-Chip Session, France*, October 1998.
- [8] J.B. Dennis. First version of a data flow procedure language. In *Programming Symposium, LNCS 19, Springer Verlag*, pages 362–376, 1974.
- [9] P. Dumont and P. Boulet. Another multidimensional synchronous dataflow: Simulating ARRAY-OL in PTOLEMY ii. Technical Report 5516, INRIA, France, March 2005. available at www.inria.fr/rrrt/rr-5516.html.
- [10] P. Feautrier. Some efficient solutions to the affine scheduling problem, i, one dimensional time. *International journal of parallel programming*, 21(5):313–348, October 1992.
- [11] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress, vol 74 of Information Processing*, pages 471–475, 1974.
- [12] 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 Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, pages 6/1–6/9, Bristol, UK, February 1996. HP Labs.
- [13] O. Labbani, J.-L. Dekeyser, P. Boulet, and E. Rutten. Introducing control in the gaspard2 data-parallel metamodel: Synchronous approach. In *Int'l Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES'05), Montego Bay, Jamaica*, October 2005.

-
- [14] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003.
- [15] C. Mauras. ALPHA : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. PhD thesis, Université de Rennes I, France, December 1989.
- [16] O. Michel, D. De Vito, and J.-P. Sansonnet. 8 1/2 : data-parallelism and data-flow. In *Proc. of the 9th International Symposium on Lucid and Intensional Programming*, 1996.
- [17] L. Morel. Efficient compilation of array iterators in LUSTRE. In *First workshop on Synchronous Languages, Applications and Programming, Grenoble*, April 2002.
- [18] I. Smarandache. *Transformations affines d'horloges: application au codesign de systèmes temps réel en utilisant les langages SIGNAL et ALPHA*. PhD thesis, Université de Rennes 1, Rennes, France, October 1998.
- [19] I.M. Smarandache, T. Gautier, and P. Le Guernic. Validation of mixed SIGNAL-ALPHA real-time systems through affine calculus on clock synchronisation constraints. In *World Congress on Formal Methods (2)*, pages 1364–1383, 1999.
- [20] J. Soula, P. Marquet, J.-L. Dekeyser, and A. Demeure. Compilation principle of a specification language dedicated to signal processing. In *6th International Conference on Parallel Computing Technologies (PaCT'2001)*, Novosibirsk, Russia, September 2001. LNCS 2127, Springer Verlag.
- [21] The WEST Team. www.lifl.fr/west/gaspard.
- [22] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, 2002.
- [23] W.W. Wadge and E.A. Ashcroft. LUCID, *the dataflow programming language*. Academic Press, 1985.



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

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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