

From high level programming model to FPGA machines

Jean-Pierre Banâtre, Dominique Lavenier, Marc Vieillot

▶ **To cite this version:**

Jean-Pierre Banâtre, Dominique Lavenier, Marc Vieillot. From high level programming model to FPGA machines. [Research Report] RR-2240, INRIA. 1994. inria-00074430

HAL Id: inria-00074430
https://inria.hal.science/inria-00074430

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

From High Level Programming Model to FPGA Machines

Jean Pierre Banâtre, Dominique Lavenier, Marc Vieillot

N° 2240

Janvier 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués



ISSN 0249-6399



From High Level Programming Model to FPGA Machines

Jean Pierre Banâtre*, Dominique Lavenier*, Marc Vieillot*

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Rapport de recherche n° 2240 — Janvier 1994 — 12 pages

Abstract: This paper presents an approach for deriving a FPGA machine from a high level parallel programming model. The model is based on the chemical reaction metaphor: the data structure is a multiset and the computation can be seen as a succession of chemical reactions consuming and producing new elements according to specific rules.

Von Neuman architecture are not suited to this programming style; we show the utility of FPGAs for deriving adapted hardware architectures. Feasibility has been demonstrated on the DEC-PRL Perle-1 board with implementation of a representative algorithm.

Key-words: Gamma, FPGA, Perle-1, architecture synthesis

(Résumé : tsvp)

- This work is funded by a DEC Perle-1 Research Initiative Contract
- to appear in the proceedings of the 2nd FPGAs Workshop for Custom Computing Machines Workshop, Napa, California, April 11-13, 1994

*{jbanatre}{lavenier}{vieillot}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Synthèse d'architecture en logique reconfigurable à partir d'un modèle de programmation de haut niveau

Résumé : Ce papier présente une approche permettant de dériver une machine en logique reconfigurable à partir d'un modèle de programmation parallèle de haut niveau. Ce modèle peut être illustré par une métaphore basée sur les réactions chimiques : la structure de données est un multi-ensemble et les calculs peuvent être représentés comme une suite de réactions consommant et produisant des éléments nouveaux en fonctions de règles spécifiques.

Les architectures de type Von Neuman se prètent mal à ce style de programmation; nous montrons l'adéquation des circuits logiques reconfigurables pour dériver des architectures adaptées. La faisabilité a été démontrée sur le système Perle-1 développé au laboratoire Parisien de DEC par le biais d'un algorithme représentatif.

Mots-clé : Gamma, logique reconfigurable, Perle-1, synthèse d'architecture

1 Introduction

Writing a correct program is difficult. It is even more difficult when the target machine is a parallel machine and no parallel programming method is provided. A computational model which allows one to describe algorithms with as few sequentiality constraints as possible is then desirable. A model without any sequentiality, called Gamma (General Abstract Model for Multiset mAnipulation), has been proposed by J.P. Banâtre and D. Le Metayer [1]. The power of Gamma is the possibility of expressing algorithms in an abstract way, without any artificial sequentiality.

The basic structure in Gamma is the multiset and a computation proceeds by nondeterministic local rewriting on that multiset. Programs based on this model are a mis-mach for Von Neuman machine. This paper will show the relevance of FPGAs for deriving an adapted hardware architecture.

The main idea suited for the Gamma paradigm is the ability of a Gamma program to be decomposed into a restrictive set of primitive programs, called *tropes* [4]. From a hardware point of view, this is an interresting property since each primitive program is well defined and induces a specific architecture.

The paper present an investigation we have performed using the Perle-1 FPGA board [2] [3] developed by the DEC Paris Research Lab. In order to show the validity of our approach, we implement manually the prime factorization algorithm.

The paper is organized as follows: section 2 gives a short presentation of the Gamma formalism; section 3 introduces the set of primitive programs (*tropes*). Section 4 presents the implementation of the prime factorization problem on the Perle-1 board.

2 The Gamma Programming Paradigm

The Gamma model can be described as a multiset transformer: the computation is a succession of applications of rules which consume elements of the multiset and produce new elements. The computation ends when no rule can be applied. The application of rules is performed in a non-deterministic way.

The basic information structuring facility is the multiset. A multiset is similar to a set except that it may contain multiple occurrences of the same element. Atomic components of multisets may be of type real, character, integer or tuple of arbitrary type.

The main feature of the model is the Γ-operator, which can be defined in the following way:

Γ (R,A) (M) =
  if $\exists\ x_1, \ldots x_n \in$ M such that R($x_1, \ldots x_n$)
    then Γ (R,A) ((M-$\{x_1, \ldots x_n\}$) $\cup$ A($x_1, \ldots x_n$))
    else M.

Operator R is called the "reaction condition"; it is a boolean function indicating under which conditions some elements of the multiset can react. The A function ("action") des-

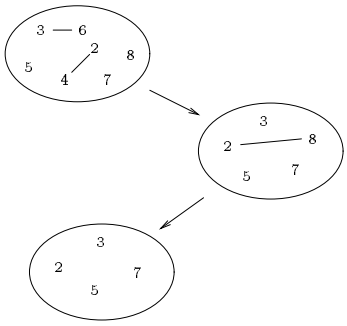
cribes the result of this reaction. We point out that if the reaction condition holds for several subsets at the same time, the choice (which is made among them) is not deterministic; if these subsets are disjoint the reactions can even take place at the same time.

Let us take one example to illustrate the programming style entailed by the Gamma-model. The sieve of Eratosthenes can be written as follows:

$$\begin{aligned}&\textbf{sieve}(\text{n}) = \Gamma\ (\text{R,A})\ (\{2, \ldots\ \text{n}\})\ \textbf{where}\\ &\qquad \text{R}(x1, x2) = \text{multiple}(x1, x2)\\ &\qquad \text{A}(x1, x2) = x2\end{aligned}$$

multiple $(x1, x2)$ is true if and only if $x1$ is a multiple of $x2$.

The following figure describes the computation of **sieve(8)**. The lines between elements indicate reactions. Of course, this is one among the possible paths leading to the stable sate.



The Gamma-model presented above is not the most general definition; actually, the Γ operator can take any number of couples (Reaction,Action), each reaction condition indicating in which case the associated action can be applied. As an example, consider the *fibonacci* computation. It can be expressed as follows:

$$\begin{aligned}&\textbf{fib}(\text{n}) = \Gamma\ (\text{R3,A3})\ (\Gamma((\text{R2,A2})(\text{R1,A1}))\{\text{n}\})\ \textbf{where}\\ &\qquad \text{R1}(x) = x > 1\\ &\qquad \text{A1}(x) = x - 1, x - 2\\ &\qquad \text{R2}(x) = x = 0\\ &\qquad \text{A2}(x) = 1\\ &\qquad \text{R3}(x1, x2) = \textbf{true}\\ &\qquad \text{A3}(x1, x2) = x1 + x2\end{aligned}$$

The initial number n is decomposed into a number of ones which are then summed-up to produce the result. The couples of Reaction/Action (R2,A2) and (R1,A1) work in parallel. Once the multiset is stable (i.e. no more reactions occur), the next Reaction/Action (R3,A3) is performed.

3 TROPES

3.1 Presentation

Tropes are a way of decomposing Gamma programs. They constitute a set of primitive program schemes, which together with just two basic combining forms provide an expressive parallel programming language. The *tropes* take the form of parametrized conditional rewrite rules in which computation proceeds in a nondeterministic local rewriting of a global multiset.

The following notation is used to denote multiset rewriting:

$$x_1, ..., x_n \rightarrow A(x_1, ..., x_n) \Leftarrow R(x_1, ..., x_n)$$

This rule can be interpreted informally as the replacement into the multiset of elements $x_1, ..., x_n$ satisfying the condition $R(x_1, ..., x_n)$ by the elements of $A(x_1, ..., x_n)$. Viewed as a single program, the result is obtained when no more rewrites can take place.

Two operators are used for combining programs: sequential composition $P_1 \circ P_2$; parallel combination $P_1 + P_2$. The intuition behind $P_1 \circ P_2$ is that the result of P_2 is passed as an argument to P_1. On the other hand, the result of $P_1 + P_2$ is obtained by performing the rewrites of P_1 and P_2 in parallel.

Five rewrite rules, called *tropes*, have been selected to provide a set of primitive programs:

- $\star$ $\mathcal{T}$ransmuter
- $\star$ $\mathcal{R}$educer
- $\star$ $\mathcal{OP}$timiser
- $\star$ $\mathcal{E}$xpander
- $\star$ $\mathcal{S}$elector

They are defined in terms of multiset rewrites. As an example, the *transmuter*, the *reducer* and the *expander* can be expressed in the following way:

$$\begin{array}{lrcl} transmuter & \mathcal{T}(C, f) & = & x \rightarrow f(x) \Leftarrow C(x) \\ reducer & \mathcal{R}(C, f) & = & x, y \rightarrow f(x, y) \Leftarrow C(x, y) \\ expander & \mathcal{E}(c, f_1, f_2) & = & x \rightarrow f_1(x), f_2(x) \Leftarrow C(x) \end{array}$$

The *transmuter* applies the same operation to all the elements of the multiset until no element satisfies the condition C. The *reducer* decreases the size of the multiset by applying a function to pairs of elements satisfying the condition C. The *expander* decomposes the elements of the multiset into a collection of basic values. The interested reader can find a complete description of the *tropes* in [4].

The examples taken in the previous section can both be expressed as a combination of *tropes*. The sieve of Eratosthenes is simply a *reducer*:

$$\text{sieve}(n) = \mathcal{R}(multiple(x, y), y)\{2, ..., n\}$$

The Fibonacci computation is a combination of three *tropes* P_1, P_2 and P_3 :

$$\text{fib}(n) = P_3 \circ (P_2 + P_1)\{n\}$$

with

$P_1 = \mathcal{E}(x > 1, x - 1, x - 2)$	*expander*
$P_2 = \mathcal{T}(x = 0, 1)$	*transmuter*
$P_3 = \mathcal{R}(true, x + y)$	*reducer*

Note that this program can also be expressed using only the sequential combination operator :

$$\text{fib}(n) = P_3 \circ P_2 \circ P_1\{n\}$$

3.2 Synthesis

From an architectural point of view the decomposition into primitive programs provides an interesting way to simplify the synthesis of Gamma programs. As the *tropes* are well defined, a hardware skeleton may be associated with each of them.

The basic procedure for synthesizing a Gamma program is first to decompose it into *tropes*. This step is done manually. In other words, the programmer has to write a Gamma program using only a set of five primitives and two operators.

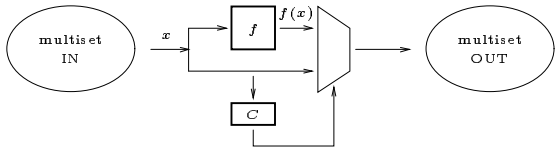
Once the *tropes* have been selected, the next step is to synthesize a corresponding architecture for each of them. Experiments have shown that the decomposition into *tropes* is not sufficient : depending on the types of the elements processed inside a *tropes*, the architecture could be rather different. An element can have the type singleton, pair, triplet, ..., etc and be composed of integer, character, etc. A pair of integers and a pair of characters have a different type. As an example, consider the *transmuter* :

$$\mathcal{T}(C, f) = x \rightarrow f(x) \Leftarrow C(x)$$

Remember that this *tropes* applies the same operation to all the elements of the multiset until no element satisfies the condition C. Two cases may be observed :

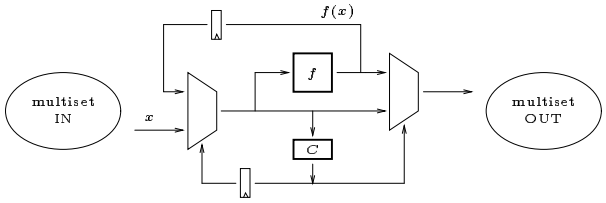
- the type of x is different from the type of $f(x)$,
- the types of x and $f(x)$ are identical.

In the first case ($\text{type}(x) \neq \text{type}(f(x))$), if an element x reacts, it will be transformed in $y = f(x)$. As the type of y is different of the type of x, we are sure that it will never react again. A possible skeleton architecture is :



The boxes f and C stand respectively for the action and the reaction. The C box drives a multiplexer which, according to the condition computation, outputs x or $f(x)$.

In the second case ($\text{type}(x) = \text{type}(f(x))$), if an element x reacts, $f(x)$ must be tested since it may react again. A possible architecture is :



This second architecture is more complex and implies a more sophisticated control.

For each *tropes*, different skeleton architectures are provided depending on the difference of the input/output types. The choice of the right *tropes* can be done automatically by an analysis of the properties of the f function.

The idea behind the refinement of the decomposition in *tropes* is to simplify hardware mechanism. It will be faster and will save FPGA resources.

The final step is to assemble the *tropes*. The combination operators $\circ$ and $+$ tell us how this assembly must be done. Actually, we focus only on the $\circ$ operator, since many programs including the $+$ operator can be rewritten using only the $\circ$ operator. Furthermore, writing $P_1 \circ P_2$ leads naturally to considering the results of P_2 as being passed as an argument to P_1. Thus, the $P_1 \circ P_2$ combination corresponds to a pipeline of the *tropes* P_1 and P_2 as shown figure 1.

It must be pointed out that the parallel operator ($+$) is more difficult to implement. Pipelining the *tropes* (using the $+$ operator) is then not possible. Data produced by a *tropes*

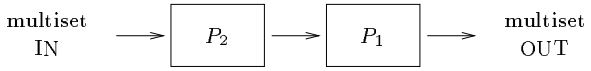


Figure 1: implementation of the sequential combination $P_1 \circ P_2$

must return to the input multiset, in order to be computed by the other *tropes*. In a first approach we discard this possibility for two main reasons:

- most of the programs can be written using only the $\circ$ operator. The expressing capability of Gamma is thus restricted but is still an interesting computational model.
- one of the main objective of this work is to investigate the PAM synthesize pprocess. The restrictive model is easily sufficient for that purpose.

3.3 Implementation

The hardware platform we use is the PRL-DEC Perle-1 board [2]. It is based on the PAM (Programmable Active Memory) [3] concept : like RAM memory module, a PAM is attached to the system bus of a host computer. The processor can write into, and read from the PAM. Being an active hardware co-processor however, the PAM processes data between write and read instructions. The specific processing is determined by the content of its *configuration memory*.

The Perle-1 board is built around a large array of bit-level configurable logic cells. This array is surrounded by local RAM banks used as a cache, a programmable clock generator and some additional logic to manage the host bus interface.

The central computational array consists of a 4×4 matrix of Xilinx XC3090 programmable gate arrays [5]. Four 32-bit wide RAM banks (1 MBytes) are provided on each side. The host bus interface is a TurboChannel interface delivering a 100 MBytes/s bandwith.

Programming Perle-1 consists of describing an architecture using an object oriented language (C++) and built-in primitive functions. More compact designs may be achieved by controlling the place and route process directly by software.

In order to manage efficiently a first implementation of *tropes* on Perle-1, we imposed some restrictions :

- only one *tropes* per FPGA,
- 16-bit datapath,
- use of the memory only if necessary.

As the matrix is a 4×4 matrix, a maximum of 16 *tropes* can be implemented. Actually, this is not really restrictive since the majority of Gamma programs can be written using very few *tropes*.

The next section presents an experiment we have performed in order to validate our approach.

4 Experiment

The goal of the experiment was twofold. First, we wanted to demonstrate that the Perle-1 board is a suitable platform to support the implementation of *tropes*. Second, we wanted to

show that a machine derived from high level specifications, such as Gamma, may have performance as good as a Von Neuman machine executing a sequential program. As a validating example, we choose the prime factorization problem.

4.1 Prime Factorization Problem

A fundamental theorem of arithmetic states that every positive integer n can be written as a product of primes and that this decomposition in unique. This fact gives a one-to-one correspondence between positive integers and a multiset of prime numbers: for example, if $n = 120 = 2^3 * 3 * 5$ the corresponding multiset is {2,2,2,3,5}.

The prime factorization Gamma program can be expressed as a combination of 6 primitive programs :

$$
\begin{array}{lcl}
\text{factor(n)} & \equiv & (F6 \circ F5 \circ F4 \circ F3 \circ F2 \circ F1)\{(2,n)_1\} \\
\text{where} & & \\
F1 & \equiv & (a,b)_1 \rightarrow (a,b)_2, (a+1,b)_1 \Leftarrow a < b \\
F2 & \equiv & (a,b)_1 \rightarrow (a,b)_2 \Leftarrow a \geq b \\
F3 & \equiv & (a,b)_2 \rightarrow\!\!\Leftarrow \neg multiple(b,a) \\
F4 & \equiv & (a,b)_2, (c,d)_2 \rightarrow (c,d)_2 \Leftarrow multiple(a,c) \\
F5 & \equiv & (a,b)_2 \rightarrow (a,b/a)_2, a \Leftarrow multiple(b,a) \\
F6 & \equiv & F3
\end{array}
$$

The initial multiset is composed of a pair, (2,n). Three different types of elements are present : pairs of type 1, noted $(a,b)_1$, pairs of type 2, noted $(a,b)_2$ and singletons.

The sequence $(F2 \circ F1)$ $\{(2,n)_1\}$ produces a multiset of pairs $(i,n)_2$ where $2 \leq i \leq n$. $F3$ removes all pairs $(i,n)_2$ which does not satisfy the condition *i does not divide n* (noted $\neg multiple(i,n)$). After the execution of $F4$, the multiset contains the pairs $(i,n)_2$ where i is a prime and i divides n. $F5$ produces the prime factor and $F6$ removes the double pairs.

Each primitive program is then associated with a *tropes*. The correspondence between the primitive programs and the *tropes* is done by analyzing the input/output types. As an example, the primitive program $F1$ is a particular *expander* which takes one pair of type $(a,b)_1$ and produces two pairs of different types ($(a,b)_1$ and $(a,b)_2$).

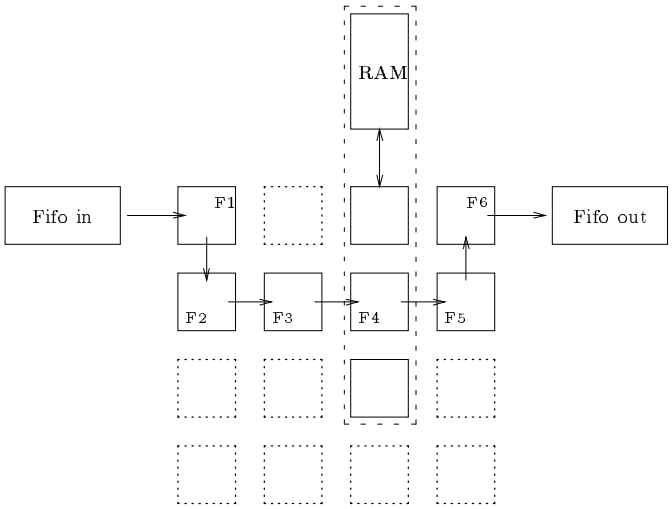


Figure 2: prime factorization implementation on Perle-1

4.2 Perle-1 Implementation

The figure 4.2 shows the implementation done on the Perle-1 board. The *tropes* are pipelined from the input FIFO to the output FIFO. One FPGA contains one *tropes* except for the *tropes* corresponding to $F4$.

Actually, this *tropes* requires to make test on two elements (say a_1 and a_2). To be efficient, it was decided when defining the *tropes* skeleton architecture to implement concurrently two tests ($C(a_1, a_2)$ and $C(a_2, a_1)$) since both reactions have to be evaluated.

In the present case, the condition has to determine if a_1 is a multiple of a_2 which is not a low cost operator in terms of CLB resources. To minimize the size of this operator, it has been implemented using a divisor operator proceeding sequentially. The division step requires two operations, a shift and a subtraction (which are done in parallel).

Implementing this *tropes* in only one FPGA is nevertheless possible if one accept to decrease performances by providing only one reaction operator. The two conditions are then evaluated sequentially.

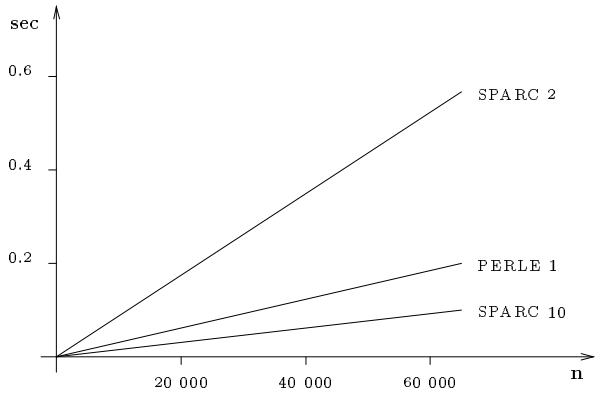
4.3 Performance Comparison

The implementation has been compared with a **similar** algorithm written in C and executed on two Sun workstations (Sparc-2 and Sparc-10). The algorithm, like the Gamma program, searches first for a prime number which divides n; then it produces the prime factors:

```
k = 0;
for(i=2; i<=n; i++){   /* F4 o F3 o F2 o F1 */
```

```
    if((n%i)==0){
      j = 1;
      while((j<=k) && ((i%tab[j]) !=0)){
        j++;
      }
      if(j > k){
        k++;
        tab[k] = i;
      }
    }
  }
  for (i=1; i<=k; i++){            /* F6 o F5 */
    v = n;
    while ((v%tab[i])==0){
      printf ("%d ",tab[i]);
      v=v/tab[i];
    }
  }
```

The diagram below shows the execution time versus the number n. The performances of the Gamma machine are situated between the performances of the two Sun Sparc Stations. This implementation demonstrates that using suitable architecture for executing high level programming language, such as Gamma, may provide performance as good as a Von Neuman machine executing a sequential program.



5 Future Work

Deriving a FPGA architecture from high level programming model using the composition of primitive programs (*tropes*) is a promising approach. Investigation for implementing the Gamma formalism on the Perle-1 FPGA platform has shown that it is realistic.

The next step is to automate this approach. Presently, specific skeleton architectures corresponding to a set of primitive Gamma programs have been defined. From these skeletons, the challenge is to synthesize efficient *tropes* and, given these *tropes*, to place them optimally on the FPGA matrix.

Software tools provided with the Perle-1 board allow direct synthesis in the same way as VLSI design. The tools support interactive placement of the CLBs, allowing the designer to control the hardware topology of the design. Experiments have shown that a *good* assignment of the CLBs with a few *judicious* routing directives may increase considerably the FPGAs possibilities: temporal performance is better and CLBs are economized.

References

[1] J.P. Banâtre and D. Le Métayer. Programming by multiset transformation. *communications ot the ACM*, 36(1):98–111, jan 1993.

[2] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McWhirter J. McCanny and E. Swartzlander, editors, *Systolic Array Processors*, pages 301–309, Pretice Hall, 1989.

[3] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories : a performance assessment. In F. Meyer auf der Heide, B. Monien, and A.L. Rosenberg, editors, *Parallel Architectures and their efficient use*, pages 119–130, Lecture notes in Computer Science, Springer-Verlag, oct 1992.

[4] C. Hankin, D. Le Metayer, and D. Sands. A Parallel Programming Style and its Algebra of Programs. *Lecture Notes in Computer Science - PARLE 93*, (694):367–378, 1993. to appear.

[5] Xilinx. *The Programmable Gate Array Data Book*. 2100 Logic Drive, San Jose, CA 95124 USA, 1992.



Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399