HAL
open science

Verification of regular architectures using ALPHA : a case study

Catherine Dezan, Patrice Quinton

To cite this version:

Catherine Dezan, Patrice Quinton. Verification of regular architectures using ALPHA : a case study. [Research Report] RR-2284, INRIA. 1994. inria-00074388

HAL Id: inria-00074388

https://inria.hal.science/inria-00074388v1

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Verification of regular architectures using ALPHA : a case study

C. Dezan, P. Quinton

N° 2284

Mai 1994

PROGRAMME 1

Rapport de recherche

ISSN 0249-6399

Verification of regular architectures using ALPHA : a case study

C. Dezan *, P. Quinton **

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

Rapport de recherche n°2284 — Mai 1994 — 22 pages

Abstract: We present a formal method for the verification of regular VLSI architectures. In our method, the behavioral specification of the chip and its implementation are first expressed in ALPHA, a functional language for the design of regular synchronous architectures. The behavioral specification is refined down to an abstract architecture description, while the implementation is simplified by induction techniques up to the same abstract architecture level. Verification is then done by matching both descriptions. This method has been successfully applied to check the correctness of a 300.000 transistor VLSI systolic chip named API69. This chip was designed to implement a string comparison algorithm with application to orthographic mispelling correction and DNA sequence comparison. We describe the ALPHA language, the formal transformations used for the verification process, and their application to the verification of the API69 chip.

Key-words: VLSI, regular architectures, hardware verification

(Résumé : tsvp)

*Université de Bretagne Occidentale, Depart. Informatique, BP 809, 29285 Brest Cedex
**IRISA, Campus de Beaulieu, 35000 Rennes. A slightly modified version of this paper appeared in Proc. ASAP'94, San Francisco, August 1994, IEEE Computer Society Press, pages 164–176. This work was partly funded by the French Coordinated Research Programs C^3 and ANM of the Ministère de la Recherche et de l'Enseignement Supérieur, and by the ESPRIT Basic Research Actions number 3280 and 6632.

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

Vérification d'architectures régulières avec ALPHA : une étude de cas

Résumé : Nous présentons une méthode formelle pour la vérification d'architectures VLSI régulières. Dans notre méthode, les spécifications comportementales du circuit et son implémentation sont d'abord exprimées en ALPHA, un langage fonctionnel pour la conception d'architectures synchrones régulières. Ces spécifications sont ensuite raffinées jusqu'à l'obtention de la description d'une architecture abstraite. L'implémentation est simplifiée par des techniques d'induction pour atteindre le même niveau d'abstraction. La vérification est alors obtenue en comparant les deux descriptions obtenues. Cette méthode a été appliquée avec succès pour vérifier la correction d'un circuit VLSI de 300.000 transistors appelé API69. Ce circuit a été conçu pour réaliser un algorithme de comparaison de chaînes de caractères avec des applications à la correction d'orthographe et la comparaison de séquences d'ADN. Nous décrivons le langage ALPHA, les transformations formelles utilisées pour la vérification, et leur application à la vérification du circuit API69.

Mots-clé : VLSI, architectures régulières, vérification de matériel

1 Introduction

Technological advance in areas of design and fabrication has increased the complexity of hardware systems. Checking the correctness of such systems is an important and difficult task, as errors may lead to costly loss of production. Many errors are due to manual design. Although automatic methods play an increasing role in VLSI design, the majority of the high level design process is still manual. Moreover, even the result of an automatic design tool might need to be checked, in order to uncover possible errors in the software tools.

Traditionally, hardware systems have been validated by means of simulation. This method is limited, as it is difficult to achieve 100% fault coverage. This is the reason why formal verification is being considered more and more often.

Formal verification is like a mathematical proof. The correctness of a hardware system is determined regardless of its input values, by considering its function rather than its behaviour, and by verifying logical properties. Formal hardware verification consists of establishing that an *implementation* satisfies a *specification*. Implementation and specification are representations of the circuit at different levels of abstraction.

Different approaches can be used for formal verification (see [Gup91] for a survey): *theorem-proving* – the relationship between a specification and implementation is treated as a theorem in logic, *model-checking* – the implementation provides a semantic model for the formula which represents the specification, *equivalence testing* – equivalence of a specification and an implementation, equivalence of finite state automata, *language containment* – the implementation is shown to be contained in the specification in the sense of the representation language. These methods depend on the specification and implementation models used: *logic* [Gord87] – propositional logic, first-order predicate logic, high-order logic, temporal logic, or *automata/language theory* [Mil91, NAP87, Ber92] – finite state automata, trace structure, process algebra.

The main limitation of verification techniques is their complexity. Although progress has been made in this area, complex examples are still beyond the capabilities of automatic verification systems. However, as pointed out by several authors, few verification techniques exploit the inherent regular nature of hardware. For parametrized descriptions, modular proofs ([HLR92]) or inductive proofs (Boyer-Moore) seem to be the most promising solutions. Among attemps to handle the verification of regular hardware, the correctness of systolic circuits has been considered by several authors (see [Hen86, ZB93] among others).

In the present paper, we present a proof methodology to check the correctness of *systolic* or more generally *regular circuits*.

The proof process is a combination of top-down *synhesis* and bottom-up *abstraction* until a common middle-point is reached. The principle of this methodology was already proposed in [Eve87] and applied to a few examples [DM90, Ueh87]. Our method takes advantage of available powerful architectural synthesis methods for systolic arrays[Qui84].

The formal representation of both the specification and the implementation of a circuit is given in ALPHA, a functional language used for the synthesis of regular architectures[VMQ91]. The ALPHA language can represent a circuit at different levels of abstraction: structural description as well as functional or temporal. The verification proceeds by doing program transformations based on the semantics of ALPHA [Mau89] thus taking advantage of the ALPHA DU CENTAUR system[DGL+91][Dez93] already used for architectural synthesis. This provides a *semi-automatic* proof process, in the sense that the designer has to select transformations whose application is automatic. The proof methodology deals directly with parametrized circuits.

The content of this paper is based on a case study. The methodology is applied to an existing 300.000 transistor systolic circuit named API69[Lav93]. The specification of the chip consists of the recurrence equations describing its operations. The implementation is the actual model of the chip at register-transfer level. By synthesis, the recurrence equations are transformed in an explicit architecture. By abstraction, the implementation is simplified by eliminating all initialization mechanisms, and some optimizations which were applied when designing the chip. The verification is completed by matching both descriptions.

The present paper is organised as follows: in section 2, we give an overview of the ALPHA language and of the basic transformations. Section 3 deals with the main transformations for architectural synthesis. Section 4 describes the transformations for abstraction. Finally, section 5 shows the application of the proof methodology to the API69 chip.

2 An introduction to the ALPHA language

ALPHA is a language invented for the synthesis of regular architecture by stepwise refinement. It is based on the model of *recurrence equations*, initially introduced by Karp, Miller and Winograd[KMW67], and extensively used later on for the synthesis of systolic arrays (see [Mol82, Qui83, Che85, DI85] among others). The ALPHA language can also be used to describe *synchronous systems*, and the-

refore, provides a natural framework for the transformation of *algorithmic specifications* into *architectures*. The principles of ALPHA have been presented in detail elsewhere[DVQS91, DGL+91, VMQ91]. The purpose of this section is to introduce the reader to the notations and principles of the language.

2.1 The basics

In the following, **Z** denotes the set of integers. In order not to introduce confusion, we denote by **integer** the usual integer type.

To help explain the language, we consider the classical example of the convolution algorithm. Given an infinite sequence $x_i, i \geq 0$, and a finite sequence of weights $w_k, 0 \leq k \leq K$, the convolution consists of computing a sequence $y_i, i \geq K$ defined by the following equation:

$$\forall i \geq K, \quad y_i = \sum_{k=0}^{K} w_k \times x_{i-k} \quad . \tag{1}$$

Let $K = 4$. This equation can be rewritten as recurrence equations as follows :

$$\begin{array}{rcl} i \geq 4 & \rightarrow & y(i) = Y(i,4) \\ i \geq 4, k = -1 & \rightarrow & Y(i,k) = 0 \\ i \geq 4, 0 \leq k \leq 4 & \rightarrow & Y(i,k) = Y(i,k-1) + w(k) * x(i-k) \quad . \end{array} \tag{2}$$

The ALPHA program presented in figure 1, describes the system of equations (2). An ALPHA program is a collection of single assignment equations. ALPHA follows the classical principles of a *structured, strongly typed functional language.*

This program has two input variables x and w, and returns a variable y. Variable x is indexed on the set $\{i | i \geq 0\}$, and can be seen as a function that maps the integral points of this domain to **integer**. Observe that this domain is a polyhedron of $\mathbf{Z}^n$. Variable w is defined on the polyhedron $\{i | 0 \leq i \leq 4\}$. It therefore represents a linear array of type **integer**. Finally, y is defined on $\{i | i \geq 4\}$. The program uses also a local variable Y, defined on the set $\{i, k | i \geq 4; -1 \leq k \leq 4\}$.

The equations defining the local variable Y and the output variable y are enclosed between the keywords **let** and **tel**.

In ALPHA, variables and expressions are functions from a set of integral points of $\mathbf{Z}^n$ called the *spatial domain* of the variable or the expression, to a set of values of a given type (**boolean**, **integer**, or **real**). ALPHA expressions are obtained by combining variables (or recursively, expressions) together with two sorts of operators : *pointwise operators* and *spatial operators.*

```
system convolution( w  : {i| 0 ≤ i ≤ 4} of integer;
                    x  : {i| i ≥ 0} of integer)
returns ( y  : {i| i ≥ 4} of integer);
var
 Y  : {i,k| i ≥ 4; −1 ≤ k ≤ 4} of integer;
let
 y =  {i| i ≥ 4} : Y.(i → i,4);
 Y =  case
      {i,k| k = −1} : 0.(i,k →);
      {i,k| 0 ≤ k ≤ 4} : Y.(i,k → i,k − 1) + w.(i,k → k) ∗ x.(i,k → i − k);
      esac;
tel
```

Figure 1: ALPHA program for the convolution

Pointwise operators. These operators are pointwise generalizations of classical arithmetic operators to ALPHA expressions. For example, given one-dimensional variables (or expressions) X and Y, the expression $\mathrm{X}+\mathrm{Y}$ represents a function defined on the intersection of the domains of X and Y whose value at index i is $\mathrm{X}_i + \mathrm{Y}_i$. Observe that pointwise operations carry out an implicit domain operation, as the resulting expression $\mathrm{X}_i + \mathrm{Y}_i$ is defined on the intersection of the domains of its operands. This is one of the key features of the language.

Spatial operators. These operators explicitly manipulate spatial domains. They are the dependence, restriction and case operators.

We call a *dependence function* an affine mapping from $\mathbf{Z}^n$ to $\mathbf{Z}^m$, $m, n \geq 0$. We denote such a function by $(i, j, \ldots \rightarrow f(i, j, \ldots))$ where f is an affine expression. For example, $(i, k \rightarrow 2i - 3k + 4)$ represents the mapping $2i - 3k + 4$ from $\mathbf{Z}^2$ to $\mathbf{Z}$. By convention, $\mathbf{Z}^0$ denotes the singleton set. A constant is a mapping from $\mathbf{Z}^0$ to its type.

The *dependence operator* combines dependence functions and expressions. Given an expression E and a dependence function d, E.d denotes the composition of functions E and d. For example, the expression $\mathrm{x}.(i \rightarrow i - 1)$ represents a "left-shift" by one of the variable x, as shown in figure 2. Let Z be this expression. The i-th element of Z is the $i-1$-th element of x. As pointwise operators, the dependence implies an operation on domains. Here, the domain of Z is the domain of x shifted to the right,

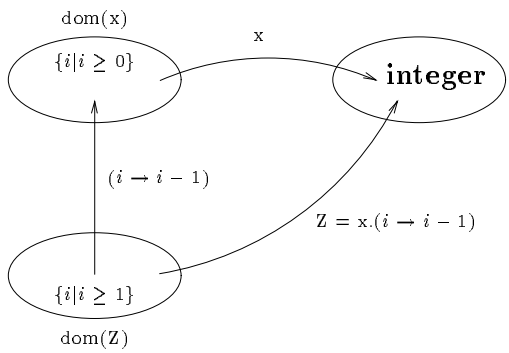


Figure 2: Illustration of dependence functions

and more generally, given an expression E and a dependency d, the domain of E.d is $\mathrm{d}^{-1}(dom(\mathrm{E}))$, the reciprocal image of the domain of E by d.

Observe that the notation $0.(i,k \rightarrow)$ extends the constant 0 to $\mathbf{Z}^2$. Similarly, $(\rightarrow k)$, where k is an integral constant, denotes a mapping from $\mathbf{Z}^0$ to $\mathbf{Z}$.

The *restriction* operator restricts the domain of an expression by means of linear constraints. Thus, $\{i,k|i \geq 3\}$:Y denotes the restriction of the function Y to the subdomain $i \geq 3$.

The **case** operator allows expressions to be conditionally defined by subdomains. It combines expressions defined on disjoint domains into a new expression, as shown by the definition of Y in program of figure 1 : depending on whether $k = -1$ or not, Y takes the value 0 or is defined recursively. To be valid, the branches of the **case** must have disjoint domains.

Observe that index names are *local* to the definitions of dependence functions and domain definitions. Thus, $(i,j \rightarrow i+j)$ is strictly equivalent to $(k,l \rightarrow k+l)$, and denotes the λ-expression $\lambda ij.(i+j)$.

2.2 Transformations

The definition of ALPHA leads to a very powerful transformation mechanism, initially aimed at providing the basic tools for the synthesis of regular arrays. In this subsection, we survey some of these transformations. They are based on the semantics of the language, which is defined in detail in [Mau89].

ALPHA follows the substitution principle: any variable can be substituted by its definition, without changing the meaning of the program. For example, substituting

Y in the definition of y in program of figure 1, gives the new definition :

$$\begin{array}{lll} y = & \{i|\, i \geq 4\} : & (\textbf{case} \\ & & \{i,k|\, k = -1\} : 0.(i,k \rightarrow); \\ & & \{i,k|\, 0 \leq k \leq 4\} : Y.(i,k \rightarrow i,k-1) + w.(i,k \rightarrow k) * x.(i,k \rightarrow i-k); \\ & & \textbf{esac};).(i \rightarrow i,4) \quad . \end{array}$$

This expression can be simplified, by distributing the outside-most restriction $\{i|\, i \geq 4\}$ inside each branch of the **case**. By simplification, the first branch disappears, its domain being empty. This leaves us with :

$$y = \quad (\{i,k|\, k = 4\} : Y.(i,k \rightarrow i,k-1) + w.(i,k \rightarrow k) * x.(i,k \rightarrow i-k)).(i \rightarrow i,4) \quad .$$

Similarly, the right-most dependency can be distributed inside the expression, and after combination with the internal dependencies, we obtain :

$$y = \quad Y.(i,k \rightarrow i,3) + w.(i,k \rightarrow 4) * x.(i,k \rightarrow i-3) \quad .$$

In fact, such a simplification can always be done. One can prove that any ALPHA expression can be rewritten as an equivalent expression, called its *normal form*, whose structure consists of one single **case** expression (see [Mau89]). The normalization process serves to simplify expressions obtained by complex transformations. It can be used, together with the substitution, to do a symbolic evaluation of an ALPHA program. It is the basis of all formal manipulation of ALPHA programs, and will be used extensively throughout the rest of this paper.

2.3 Change of basis

A *change of basis* can be applied to the index space of any local variable, using a straightforward syntactic transformation of the equations [VMQ91]. The change of basis transformation is used for *space-time reindexing*, which is the most fundamental transformation of the synthesis of regular array. The idea behind this transformation is to reindex the variables of a program in such a way that one dimension of the new index space can be interpreted as the *time*, while the remaining indexes represent a processor number (see figure 3).

This transformation is valid if the change of basis is done with an *integral unimodular*[1] matrix [Mau89]. A change of basis by a unimodular function P (expressed in dependence form), whose inverse is Q, on the variable Z defined on a domain Dom is achieved as follows:

[1] One nonsingular matrix M is said to be unimodular if $det(M) = \pm 1$

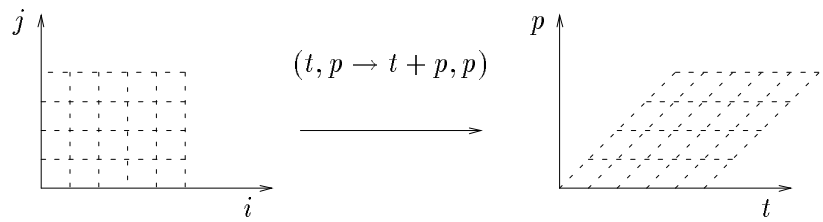


Figure 3: Example of change of basis

- the definition of Z = exp is replaced by Z = P(Dom) : $exp.Q$, where P(Dom) represents the image by P of the domain Dom,
- each occurence of Z in the right-hand side of equation is replaced by Z.P.

Actually, the change of basis can be extended to the case when P has a left inverse Q, i.e. $QP = \mathrm{I}$. It suffices to replace in the above transformation the matrix P^{-1} by Q (see [HL94]).

2.4 ALPHA array notation

The notation used so far for ALPHA program is functional, and suited to the understanding of the semantics of ALPHA. However, for practical purposes, it is easier to understand a definition $X = exp$ as a quantified statement of the form $X[i] = exp[f(i)]$. We call such a notation the *array notation* of ALPHA. The translation of a normalized ALPHA equation to its array notation is straightforward. For example, the array notation for the convolution is :

```
Y[i,k] = case
        k = -1 : 0;
        0 ≤ k ≤ 4: Y[i,k-1] + w[k] * x[i-k];
esac;
y[i]= Y[i,n];
```

Suppose now that we substitute Y in the definition of y. The array notation for this new program is:

```
y[i]=(lambda ik.
      Y[i,k] = case
            k = -1 : 0;
            0 ≤ k ≤ 4: Y[i,k-1] + w[k] * x[i-k];
      esac;)[i,n];   .
```

3 Synthesis transformations

As explained in the introduction, the top-down part of our verification methodology consists of transforming the functional specification of the chip into an architecture, by means of synthesis transformations. In this section, we present these transformations. Subsection 3.1 summarize the well known space-time transformation technique. In subsections 3.2 and 3.3, we describe two additional transformations needed to synthesize hardware.

3.1 Time-space mapping

The synthesis procedure is based on the following remarks. First, the behaviour of a synchronous architecture can be represented by a set of recurrence equations, (and therefore, as an ALPHA program,) where one index, say t, is the time, and other indices are a multidimensional processor number. For example, the well known unidirectional systolic array for convolution can be described by means of the program of figure 4. The architecture can be deduced from this program by an *interpretation* of the indexes : t means the time, and p means the processor number. With such an interpretation, a dependence function $(t, p \rightarrow t-1, p)$ can be translated into a register, and a dependence function $(t, p \rightarrow t, p-1)$ represents a connection between processor p and processor $p-1$.

The goal of the synthesis is to transform the initial description of 1 into that of figure 4. To do so, the following steps are (see [VMQ91] for details):

- apply to Y a change of basis which maps (i, k) to $(t = i + k, p = k)$. The determination of this transformation is done by solving a linear programming problem,
- pipeline variable X in such a way that it circulates in the same direction as Y.

```
system convolution( w  : {i| 0 ≤ i ≤ 4} of integer;
                    x  : {i| i ≥ 0} of integer)
returns ( y  : {i| i ≥ 4} of integer);
var
 X, Y  : {t, p|t ≥ 0, 0 ≤ p ≤ 4} of integer ;
let
    y[i] =  {i| i ≥ 4} : Y[i + 4, 4];
  Y[t, p] =  case
             p = −1 : 0
             0 ≤ p ≤ 4 : Y[t − 1, p − 1] + w[p] ∗ X[t, p];
             esac;
  X[t, p] =  case
             p = −1 : 0;
             0 ≤ p ≤ 4 : X[t − 2, p − 1];
             esac;
tel
```

Figure 4: ALPHA program for the convolution

3.2 Clock refinement

It is often the case that the implementation of a systolic array uses a two-level clock : the systolic cycle is itself subdivided into smaller cycles, during which elementary instructions are performed. In order to model such an operation, one can refine the interpretation of the ALPHA program by allowing two different indexes to represent the time : t for the systolic clock, and k, say, to represent the instruction number within a systolic cycle. With such an interpretation, the actual clock number of an instruction is given by clock $= lt+k$ where l is the number of instructions by systolic cycle.

Formally, the clock refinement transformation can be achieved by means of a change of basis. Consider the following program fragment

$$\begin{array}{l} \text{X, Y: } \{t|t \geq 0\} \ \ \textbf{of} \text{ boolean;} \\ \quad \text{X}[t] = \text{X}_1[t] + \text{X}_2[t]; \\ \quad \text{Y}[t] = \text{X}[t] + \text{X}_3[t]; \quad , \end{array}$$

whose hardware intepretation is shown in figure 5a. In order to map X to the first cycle $k = 0$ and Y to the second cycle $k = 1$ of a refined clocking scheme, we apply

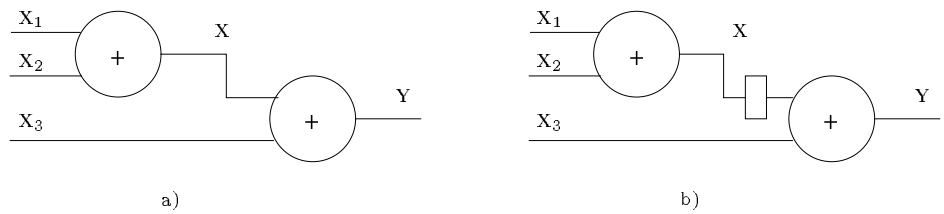


Figure 5: Illustration of clock refinement

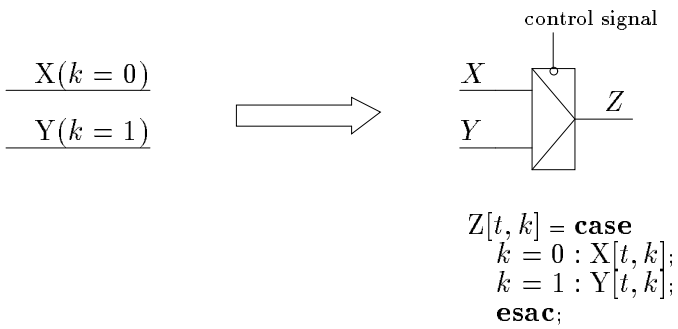


Figure 6: Merging to variables for hardware sharing

the change of basis $P_1 = (t \to t, 0)$ to X, and $P_2 = (t \to t, 1)$ to Y. The common left inverse of P_1 and P_2 is $Q = (t, k \to t)$. Applying these transformations leads to the new program

$$\begin{array}{l} \mathrm{X}\colon \{t,k|t \geq 0; k = 0\} \ \ \textbf{of}\ \text{boolean}; \\ \mathrm{Y}\colon \{t,k|t \geq 0; k = 1\} \ \ \textbf{of}\ \text{boolean}; \\ \quad \mathrm{X}[t,k] = \mathrm{X}_1[t] + \mathrm{X}_2[t]\ ; \\ \quad \mathrm{Y}[t,k] = \mathrm{X}[t,0] + \mathrm{X}_3[t]; \quad , \end{array}$$

shown in figure 5b. Notice the register introduced between X and Y. This is because $clock_{\mathrm{Y}} = clock_{\mathrm{X}} + 1$.

3.3 Hardware sharing

Hardware sharing is also a very useful hardware synthesis transformation. It can be formally represented in ALPHA as follows. Let X and Y be two variables defined on disjoint domains d_{X} and d_{Y} respectively. Assume that $d_X \cap d_Y = \emptyset$. Then Z = **case** X ; Y **esac** is a valid definition. By adding this definition to a program, and

by replacing each occurrence of X by d_X : Z, and each occurrence of Y by d_Y : Z, one obtains a new equivalent program. This transformation, when applied to an ALPHA program being interpreted in terms of hardware, models hardware sharing, as illustrated in figure 6. Here, the **case** expression is interpreted as a multiplexer.

4 Abstraction transformations

The second part of our verification methodology consists of simplifying the description of the actual chip, in order to obtain a representation at the same level as the synthesized one. As we shall see in section 5, the main difficulty encountered in our case study was to simplify the equations describing the initialization process of the chip. During this phase, registers are loaded with values, which are used as constants during the subsequent calculations. A natural way to simplify the chip was to prove that these values are constant, and replace them in the program – an technique similar to constant propagation in optimizing compilers. As shown here, this can be done by applying a simple induction scheme, which, thanks to the properties of ALPHA, leads in our particular case to a very simple verification process.

In this section, we consider the proof of a predicate P which involves only unidimensional variables. Let $\mathbf{N}$ denote the set of natural numbers. To prove P, one proceeds as follows :

1. prove $P(0)$ (base case),
2. prove that $P(n) \Rightarrow P(n+1), \forall n \in \mathbf{N}$ (inductive step).

To express this in ALPHA, let us define boolean variables P_{base}, and P_{next} respectively representing $P(0)$, $P(n+1)$. These variables are defined on the domain $\{n | n \geq 0\}$, and their ALPHA definition is

$$\begin{array}{l} P_{\text{base}}[n] = n \geq 0 : P[0] \\ P_{\text{next}}[n] = n \geq 0 : P[n+1] \quad . \end{array}$$

To prove P, one has to prove that P_{base} and $P \Rightarrow P_{\text{next}}$ are true for all n.

Consider now the following example. Let X be a variable defined by

$$\begin{array}{lcl} \mathrm{X}[t] & = & \textbf{case} \\ & & t = 0 : \mathrm{x0}; \\ & & t \geq 1 : \mathrm{X}[t-1]; \\ & & \textbf{esac}; \end{array}$$

and suppose that we want to prove that for $t \geq 0$, $\mathrm{X}(t) = \mathrm{x0}$. The property to prove can be written in ALPHA :

$$\mathrm{P}[t] = t \geq 0 : (\mathrm{X}[t] = \mathrm{x0}).$$

We build P_{base} and P_{next} :

$$\begin{array}{lll} P_{\text{base}}[t] = & t \geq 0: & \mathrm{P}[0] \\ P_{\text{next}}[t] = & t \geq 0: & \mathrm{P}[t+1] \end{array}$$

By substituting P by its definition in P_{base} and in P_{next}, we obtain :

$$\begin{array}{lll} \mathrm{P}_{\text{base}}[t] = & t \geq 0: & (\mathrm{X}[0] = \mathrm{x0}), \\ \mathrm{P}_{\text{next}}[t] = & t \geq 0: & (\mathrm{X}[t+1] = \mathrm{x0}). \end{array}$$

To prove P_{base}, let us substitute X by its definition. The result is

$$\begin{array}{lll} P_{\text{base}}[t] = & t \geq 0: & ((\textbf{lambda}\ t. \\ & & \textbf{case} \\ & & t = 0 : \mathrm{x0}; \\ & & t \geq 1 : \mathrm{X}[t-1]; \\ & & \textbf{esac};\ )[0] = \mathrm{x0}); \end{array}$$

After normalization, we get

$$P_{\text{base}}[\mathrm{t}] = \quad t \geq 0: \quad (\mathrm{x0} = \mathrm{x0});$$

which proves the base case.

To prove P_{next}, let us substitute again X by its definition in the definition of P_{next}. This leads to :

$$\begin{array}{lll} P_{\text{next}}[t] = & t \geq 0: & ((\textbf{lambda}\ t. \\ & & \textbf{case} \\ & & t = 0 : \mathrm{x0}; \\ & & t \geq 1 : \mathrm{X}[t-1]; \\ & & \textbf{esac};\ )[t+1] = \mathrm{x0}); \end{array}$$

It turns out that, again, normalization provides the desired result. Indeed, distributing the dependence $[t+1]$ gives :

$$
\begin{array}{lll}
P_{\text{next}}[t] = & t \geq 0: & ((\textbf{case} \\
 & & t = -1 : \text{x0}; \\
 & & t \geq 0 : \text{X}[t]; \\
 & & \textbf{esac};) = \text{x0});
\end{array}
$$

By distributing the pointwise = operator, we obtain

$$
\begin{array}{lll}
P_{\text{next}}[t] = & t \geq 0: & (\textbf{case} \\
 & & t = -1 : (\text{x0} = \text{x0}); \\
 & & t \geq 0 : (\text{X}[t] = \text{x0}); \\
 & & \textbf{esac};);
\end{array}
$$

Finally, as $t \geq 0 \wedge t = -1 = \emptyset$, we get

$$
P_{\text{next}}[t] = \quad t \geq 0: \quad (\text{X}[t] = \text{x0});
$$

This is exactly the definition of P, which proves the induction step.

5 Application to API69

In this section, we describe the application of our methodology to the verification of a special purpose chip. In subsection 5.1, the chip is presented, and the overall structure of the verification is described. Subsections 5.2 and 5.3 show respectively the synthesis and abstraction part.

5.1 Presentation of the case study

The chip, named API69 [Lav93], is a VLSI chip dedicated to string comparison by means of dynamic programming techniques. Given two character strings $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$, the chip computes the distance between X and Y given by the minimum number of *substitutions*, *deletions*, *insertions*, and *transpositions* which are necessary to transform X into Y. Let $\wedge$ represent the empty character. It has been shown in [LOW75] that $D(i,j)$, the distance between the i first characters of X and the j first characters of Y, is obtained using the following recurrence (the transposition case is not represented here, for the sake of simplicity:)

$$
D(i,j) = Min \begin{cases} D(i-1,j-1) + d(x_i, y_j) \\ D(i-1,j) + d(\wedge, y_j) \\ D(i,j-1) + d(x_i, \wedge) \quad , \end{cases} \tag{3}
$$

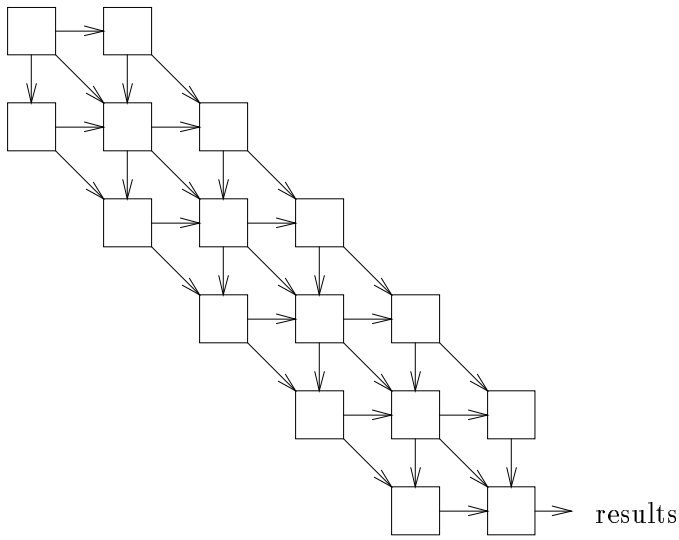


Figure 7: Array structure

where $d(x_i, \wedge)$ represents the cost of deleting x_i, $d(\wedge, y_j)$ the cost of inserting y_j, and $d(x_i, y_j)$ the cost of substituting x_i by y_j.

This algorithm can be mapped onto a rectangular systolic array of dimension $n \times m$ where each processor is capable of performing one elementary computation $D(i, j)$. In [LAV89] and [SCHA90], it has been shown that in practice three diagonals of processors are sufficient to provide good results. The array is thus organized as shown in figure 7.

The API69 chip is based on this structure. However, it contains several optimizations which make its actual implementation far from trivial. These optimizations concern :

- the internal structure of processors,
- the configuration process during which each processor is made particular,
- the mapping of interconnections between processors, in order to minimize transmitted data.

The overall verification process of API69 is conducted as follows. First, the equations describing the systolic array in figure 7 at a functional level are translated into

```
system      initial_spec (ki,ko: integer;
            ts : {t,i | t ≥ 0; 6 ≥ i; i ≥ 1} of integer)
            returns (s : {t,i,j | t ≥ 0; 6 = j; 6 = i} of integer);
var
            D : {t,i,j | t ≥ 0; i ≥ 0; j ≥ 0; 6 ≥ i; j + 2 ≥ i; i + 2 ≥ j; 6 ≥ j} of integer;
let
D[t,i,j] =  case
            t ≥ 0; i = 0; j = 0 : 0;
            t ≥ 0; 1 = i; j = 0 : ki;
            t ≥ 0; 2 = i; j = 0 : ki + ki;
            t ≥ 0; i = 0; 1 = j : ko;
            t ≥ 0; i = 0; 2 = j : ko + ko;
            t ≥ 0; 6 ≥ i; i ≥ 3; i = j + 2 : infty;
            t ≥ 0; 4 ≥ i; i ≥ 1; i + 2 = j : infty;
            t ≥ 0; j + 1 ≥ i; j ≥ 1; i ≥ 1; i + 1 ≥ j; 6 ≥ i; 6 ≥ j :
            min ( D[t − 1, i, j − 1] + ko ,
            min ( D[t − 2, i − 1, j − 1] + ts[t − 1, i] ,
            D[t − 1, i − 1, j] + ki ) );
            esac;
s[t,i,j] =  D[t, 6, 6];
tel;
```

Figure 8: Initial specification of API69

ALPHA (we skipped the mapping of the recurrence equations to the systolic array, since, in this case, it is an obvious transformation). Second, the actual chip is also described using ALPHA.

From these specifications, we proceed in the following way :

- by synthesis, we refine the functional description in order to come closer to the implementation by detailing the I/O activity of each processor. We also reduce the number of processor interconnections in order to get closer to the implementation which does not contain diagonal links.
- by abstraction, we eliminate the *masking* mechanism present in the processors to support their initialization.
- We then show that the synthesized representation contains the implementation.

We now detail these steps.

5.2 Synthesis of API69 specification

The initial specification shown in figure 8 is an almost direct and straightforward translation of the systolic representation of algorithm 3. In the following ALPHA program, *ko*, *ki*, and *ts* represent respectively the cost of deletion, of insertion, and of substitution. Deletion and insertion are assumed to have a cost independent of the character considered. This representation is given in term of "systolic time". The synthesis process consists essentially of two operations. First, variables are "allocated" to a subcycles of the systolic cycle, as explained in 3.2. Second, the connections between processors represented by the dependences between $\mathrm{D}[t,i,j]$ and $\mathrm{D}[t-1,i,j-1]$, $\mathrm{D}[t-2,i-1,j-1]$ or $\mathrm{D}[t-1,i-1,j]$ are mapped to a minimum number of physical links. If we affect the diagonal dependence and the horizontal one to different instructions of the clock, we can use a single physical link for both communications. This is done through hardware sharing, as explained in 3.3.

5.3 Abstraction of the chip

The physical description of the chip is at the register transfer level. Notice that this description also contains the specification of the I/O behavior of the chip, that is to say, when data are fed into the chip, and results are output.

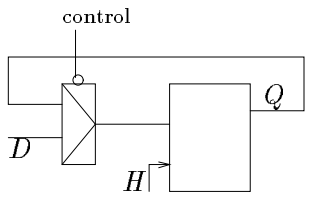
The implementation of the API69 chip uses identical processors, which are configured during an initialization phase. Control signals are set in each processor. They activate where needed using a *masking* mechanism which enables a given processor to replace on the fly a datum communicated by a neighbouring processor with the contents of a register. This implementation techniques allows a very regular hardware design to be obtained, in spite of the numerous special cases caused by the boundary conditions of the recurrences.

The *mask* mechanism is implemented by a static register that can be modelled as shown in fig. 9, and which corresponds to the ALPHA equation

$$\mathrm{Q}[t] = (\textbf{if}\ \text{control}\ \textbf{then}\ \mathrm{D}\ \textbf{else}\ \mathrm{Q})[t-1] \quad . \tag{4}$$

Here, control is a boolean signal *true* only during the first tick of the clock. The ALPHA expression of the control is :

$$\begin{array}{lcl} \text{control}[t] & = & \textbf{case} \\ & & t = 1 : \text{true}; \\ & & t \geq 2 : \text{false}; \\ & & \textbf{esac}; \end{array}$$



$\mathrm{Q}[t] = (\textbf{if}\ \mathrm{control}\ \textbf{then}\ \mathrm{D}\ \textbf{else}\ \mathrm{Q})[t-1]$

Figure 9: Model of a static register and its ALPHA representation

Equation (4) can be reexpressed by substituting the variable control. In this way, the output Q is now defined by a temporal recurrence :

$$\begin{array}{lcl} \mathrm{Q}[t] & = & \textbf{case} \\ & & t = 1 : \mathrm{D}[t-1]; \\ & & t \geq 2 : \mathrm{Q}[t-1]; \\ & & \textbf{esac}; \end{array}$$

By substituting D by its definition once more, we obtain a value of one point, the initial value of D which is a constant value cst. One can now prove by induction that Q is equal to cst when $\{t \geq 1\}$ using the method presented in section 4.

5.4 Meet-in-the-middle comparison

Synthesis and abstraction both result in ALPHA programs at a same level of expression, in which :

- processors are described as a function from input to output ;
- all processors are described by the same functional expression; the particular behavior of border processors is handled by internal variables loaded during initialization ;
- the whole systolic array is described in a structural way: interconnections between processors are explicitly described.

The final description of the API69 is shown in appendix A.

6 Conclusion

We have presented a verification methodology for regular circuits based on program transformation of ALPHA programs. Correctness of the regular circuit is established

by showing that the ALPHA description of the chip is equivalent to the ALPHA description of its behavior in term of recurrence equations. To do so, the functional description is transformed into an architecture, by means of top down synthesis, and the implementation is abstracted up to the same representation level, by means of induction. We have shown the application of this method to the verification of the API69 chip, a 300.000 transistors IC for string comparison.

The effectiveness of our methodology relies upon the strong formal properties of the ALPHA language on one hand, and upon its underlying model, systems of recurrence equations, on the other hand. The properties of ALPHA, initially oriented towards the synthesis of systolic array, allow parameterized regular descriptions to be treated in a very concise way, thanks to the properties of polyhedral domains and affine dependence functions it is based on. The systematic use of substitution and normalization help simplifying expressions obtained throughout the verification process. On the other hand, systems of recurrence equations is a powerful model which is well suited to the representation of regular architectures.

The cooperation of top-down synthesis and bottom-up abstraction is also a key point of the method. The synthesis part allows the gap between the functional specification and the abstract architecture to be filled in a very efficient way. On the other hand, abstraction by induction is the only way to deal with the peculiarities of the physical implementation.

The proof by induction which has been described in this paper for one dimensional variables, can certainly be generalized to more general cases. Moreover, the structure of the proof is obviously related to the domains of the variables involved in the predicate to be proven, and there is hope that this structure can be found in an automatic way.

References

[Che85] M.C. Chen. The generation of a class of multipliers: A synthesis approach to the design of highly parallel algorithms in VLSI. In *IEEE Int. Conf. on Computer Design: VLSI in Computers*, pages 116–121, Oct. 7-10 1985.

[DGV+91] C. Dezan, E. Gautrin, H. Le Verge, P. Quinton, and Y. Saouter. Synthesis of systolic arrays by equation transformations. In *ASAP'91*, Barcelona, Spain, September 1991. IEEE.

[DI85] J.M. Delosme and I.C.F. Ipsen. An illustration of a methodology for the construction of efficient systolic architectures in VLSI. In *Internatio-*

nal Symposium on VLSI Technology, Systems and Applications, Taipei, Taiwan, R.O.C., pages 268–273, May 1985.

[DVQS91] C. Dezan, H. Le Verge, P. Quinton, and Y. Saouter. The ALPHA DU CENTAUR environment. In P. Quinton and Y. Robert, editors, *International Workshop Algorithms and Parallel VLSI Architectures II*, pages 325–334, Bonas, France, June 1991. North-Holland.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.

[Mau89] C. Mauras. Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. Thèse de l'Université de Rennes 1, IFSIC, December 1989.

[Mol82] D.I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Transactions on Computers*, C-31(11), November 1982.

[VMQ91] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[Ber92] G. Berry. A hardware implementation of pure esterel. In *Academy Proceedings in Engineering Sciences*, volume 17, pages 95 –130, Sadhana, March !992.

[Dez93] Catherine Dezan. *Generation automatique de circuits avec ALPHA du CENTAUR*. PhD thesis, Universite de Rennes I, February 93.

[DGL+91] C. Dezan, E. Gautrin, H. Leverge, P. Quinton, and Y. Saouter. Synthesis of systolic arrays by equation transformations. In *ASAP'91*, Barcelona, Spain, 1991. IEEE.

[DM90] Steven D.Johnson and Robert M.Wehrmeister. On the interplay of synthesis and verification, experiments with the fm8501 processor description. In *Formal VLSI Specification and Synthesis: VLSI Design Methodes I (IFIP)*. Elsevier Science publishers BV (North-Holland), 1990.

[Duf91] David Duffy. *Principles of automated theorem proving.* Wiley, 1991.

[Eve87] Hans Eveking. Verification, synthesis and correctness-preserving transformations cooperative approaches to correct hardware design. In D. Borrionne, editor, *From HDL Descriptions to Guaranted Correct Circuit Designs (IFIP)*. Elsevier Science Publishers BV (North-Holland), 1987.

[Gord87] M.J.C. Gordon HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subramanyan, editors. *VLSI Specification, Verification and Synthesis*, pages 73-128. Kluwer Academic Publishers, 1987.

[Gup91] Aarti Gupta. Formal Hardware Verification Methods: A survey. Internal report of Carnegie Mellon University, CMU-CS-91-193, October 1991.

[HL94] P. Quinton H. Le Verge. Recurrences on lattice polyhedra and their applications to the synthesis of systolic arrays. Publication interne, IRISA, Irisa, Campus de Beaulieu, 35042 Rennes Cedex, France, 1994. To appear.

[Hen86] M. Hennessy. Proving systolic systems correct. *ACM Toplas*, 8(3):344–387, July 1986.

[KL78] H.T Kung and C.E Leiserson. Systolic arrays for vlsi. In *Sparse Matrix Proc*, pages 256–282. Society for Industrial and Applied Mathematics, 1978.

[LAV89] D. Lavenier, " MicMacs : un réseau systolique linéaire programmable pour le traitement de chaînes de caractères," *Thèse de l'université de Rennes 1,* Juin 1989.

[Lav93] Dominique Lavenier. An Integrated 2D Systolic Array for Spelling Correction. *INTEGRATION: the VLSI journal*, 15:97–111, August 1993.

[LMQ90] H. Le Verge, C. Mauras, and P. Quinton. A language-oriented approach to the design of systolic chips. In *International Workshop on Algorithms and Parallel VLSI Architectures*, Pont-à-Mousson, June 1990. To appear in the Journal of VLSI Signal Processing, 1991.

[LOW75] R. Lowrance, R. A. Wagner, "An extension of the string to string correction problem," *J. Assoc. Comput. Mach.,* vol. 22, n° 2, pp. 177-183, 1975.

[Mau89] Christophe Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèle synchrones.* PhD thesis, Université de Rennes I, 1989.

[Mil91] G. J. Milne. The formal description and verification of hardware timing. *IEEE Transaction on Computers*, 40(7), July 1991.

[Qui83] P. Quinton. The systematic design of systolic arrays. Technical Report 193, Publication Interne IRISA, Avril 1983.

[Qui84] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. IEEE 11-th Int. Sym. on Computer Architecture*, pages 208–214, Ann Arbor, Mi, USA, 1984.

[NAP87] N.Halbwachs, A.Lonchampt, and D. Pilaud. Describing and designing circuits by means of synchronous declarative language. In D.Borrione, editor, *From HDL Descriptions to Garanteed Correct Circuits Designs*, pages 255–268. North-Holland, 1987.

[HLR92] N.Halbwachs, F.Lagnier, and C.Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 1992.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley Interscience serie in Discrete Mathematics, john wiley and sons edition, 1986.

[SCHA90] Jean Luc Chambarg. *Une machine systolique adaptée à la correction de chaîne de caractères : application aux adresses postales.* PhD thesis, Université de Rennes I, 1990.

[Ueh87] Takao Uehara. Proofs and synthesis are cooperative approaches for correct circuit designs. In D.Borrionne, editor, *From HDL Description to Guaranted Correct Circuit Designs (IFIP).* Elsevier Science Publishers BV (North-Holland), 1987.

[ZB93] Z. Zhou and W. Burleson. Formal descriptions, semantics and verification of vlsi array processors. In L. Dadda and B. Wah, editors, *ASAP'93*, pages 321–332, Venise, October 1993. IEEE.

A The final description of API69

The following is the common ALPHA description reached by synthesis and abstraction. The INP equation describes links between processors. The dependences used assume that horizontal and vertical transfer are useful. These links are configurable in the sense that a single variable INP transmit alternatively data from horizontal or vertical processor. The input variables DD, DV and DH represent the distances computed by the diagonal, vertical or horizontal neighbour processor. Their values are particular for border processors. For each processor, TS1 represents the access to the *substitution* coefficient which is an input value expressed in TS. Computation of each processor can be expressed in one single equation. This is represented by D equation. The outputs of processor expressed by OutP are the result of computation when they transfer D otherwise the diagonal data is transmited through INP.

system api69 (ki,ko : **integer**; ts : $\{t, i \mid t \geq 0; 6 \geq i; i \geq 1\}$ of **integer**)
returns (s : $\{t, i, j \mid t \geq 0; 6 = j; 6 = i\}$ of **integer**);
var
DD : $\{t, i, j, k \mid t \geq 0; k = 0; i + 1 \geq j; j \geq 1; i \geq 1; j + 1 \geq i; 6 \geq j; 6 \geq i\}$ of **integer**;
DV : $\{t, i, j, k \mid t \geq 0; 2 = k; j \geq 1; i \geq 1; i + 1 \geq j; j + 1 \geq i; 6 \geq i; 6 \geq j\}$ of **integer**;
DH : $\{t, i, j, k \mid t \geq 0; 1 = k; i + 1 \geq j; j \geq 1; i \geq 1; j + 1 \geq i; 6 \geq j; 6 \geq i\}$ of **integer**;
TS1 : $\{t, i, j, k \mid t \geq 0; j + 1 \geq i; k = 0; j \geq 1; i \geq 1; i + 1 \geq j; 6 \geq i; 6 \geq j\}$ of **integer**;
INP : $\{t, i, j, k \mid t \geq 0; i \geq 1; k \geq 0; j \geq 1; j + 1 \geq i; 2 \geq k; i + 2 \geq j; 6 \geq j; 6 \geq i\}$ of **integer**;
D : $\{t, i, j, k \mid t \geq 0; 3 = k; j \geq 1; i \geq 1; i + 1 \geq j; j + 1 \geq i; 6 \geq i; 6 \geq j\}$ of **integer**;
OutP : $\{t, i, j, k \mid t \geq 0; i \geq 1; k \geq 0; j \geq 1; j + 1 \geq i; 2 \geq k; i + 2 \geq j; 6 \geq j; 6 \geq i\}$ of **integer**;
TS : $\{t, i, k | t \geq 0; i \geq 1; k \geq 0; 2 \geq k; 6 >= i\}$of **integer**; **let**
–processor interconnexion
INP$[t, i, j, k]$ = **case**
$t \geq 0; 1 = k; j \geq 2; j \geq i; i + 1 \geq j; 6 \geq j$: OutP$[t, i, j - 1, k]$;
$t \geq 0; 1 = k; 4 \geq i; i \geq 1; i + 2 = j$: OutP$[t, i, j - 1, k]$;
$t \geq 0; 2 = k; i \geq 3; 6 \geq i; j + 1 = i$: OutP$[t, i - 1, j, k]$;
$t \geq 0; 2 = k; i \geq 2; i + 1 \geq j; j \geq i; 6 \geq j$: OutP$[t, i - 1, j, k]$;
$t \geq 0; 6 \geq i; i \geq 2; k = 0; i = j + 1$: OutP$[t, i - 1, j, k]$;

```
        t ≥ 0; i ≥ 2; 6 ≥ i; k = 0; j = i : OutP[t, i − 1, j, k];
esac;
–inputs of processors
DD[t, i, j, k] = case
        t ≥ 0; 3 = k; 1 = j; 1 = i : 0;
        t ≥ 0; 3 = k; 1 = j; 2 = i : ki;
        t ≥ 0; 3 = k; 2 = j; 1 = i: ko;
        t ≥ 0; 3 = k; 6 ≥ i; i ≥ 2; i = j + 1: INP[t, i, j, k − 1];
        t ≥ 0; 3 = k; 6 ≥ i; i ≥ 2; i = j: INP[t, i, j, k − 1];
        t ≥ 0; 3 = k; i ≥ 2; 5 ≥ i; j = i + 1: INP[t, i, j, k − 1];
esac;
DV[t, i, j, k] = case
        t ≥ 0; 2 = k; 1 = j; 1 = i : ki;
        t ≥ 0; 2 = k; 1 = j; 2 = i: 2 * ki;
        t ≥ 0; 2 = k; 2 = j; 1 = i: INP[t, i, j, k − 1];
        t ≥ 0; 2 = k; i ≥ 2; 6 ≥ i; j + 1 = i: infty;
        t ≥ 0; 2 = k; i ≥ 2; 5 ≥ i; j = i + 1 : INP[t, i, j, k − 1];
        t ≥ 0; 2 = k; i ≥ 2; 6 ≥ i; j = i: INP[t, i, j, k − 1];
esac;
DH[t, i, j, k] = case
        t ≥ 0; 1 = k; 1 = j; 1 = i: ko;
        t ≥ 0; 1 = k; 1 = j; 2 = i: INP[t, i, j, k − 1];
        t ≥ 0; 1 = k; 2 = j; 1 = i: 2 * ko;
        t ≥ 0; 1 = k; 6 ≥ i; i ≥ 2; i = j + 1: INP[t, i, j, k − 1];
        t ≥ 0; 1 = k; i ≥ 2; 6 ≥ i; j = i : INP[t, i, j, k − 1];
        t ≥ 0; 1 = k; 5 ≥ i; i ≥ 2; i + 1 = j: infty;
esac;
TS[t, i, k] = ts[i, k]
TS1[t, i, j, k] = case
        t ≥ 0; 3 = k; 6 ≥ i; i ≥ 1; j + 1 = i : TS[t, i, k − 3];
        t ≥ 0; 3 = k; 6 ≥ i; i ≥ 1; i = j: TS[t, i, k − 2];
        t ≥ 0; 3 = k; 5 ≥ i; i ≥ 1; i + 1 = j: TS[t, i, k − 1];
esac;
–computation of processors
D[t, i, j, k] = min ( DV[t, i, j, k − 1] + ko ,
        min ( DH[t, i, j, k − 2] + ki ,
        DD[t, i, j, k − 3] + TS1[t, i, j, k − 3] ) );
```

–outputs of processors
$\mathrm{OutP}[t,i,j,k] =$ **case**
$t \geq 1; j >= i; i \geq 1; k = 0; 5 \geq i; i+1 \geq j$: $\mathrm{D}[t,i,j,k]$;
$t \geq 0; 2 = k; 2 = j; 1 = i$: $\mathrm{INP}[t,i,j,k-1]$;
$t \geq 0; 2 = k; i \geq 2; 5 \geq i; j = i+1$: $\mathrm{INP}[t,i,j,k-1]$;
$t \geq 0; 2 = k; i \geq 2; 5 \geq i; j = i$: $\mathrm{INP}[t,i,j,k-1]$;
$t \geq 0; 2 = k; i \geq 1; 4 >= i; j = i+2$: $\mathrm{INP}[t,i,j,k-1]$;
$t \geq 1; 1 = k; j \geq 1; i+1 \geq j; i \geq 1; j+1 \geq i; 5 \geq j$: $\mathrm{D}[t,i,j,1]$;
esac;
–output of the array
$\mathrm{s}[t,i,j] = \mathrm{D}[t,6,6,0]$
tel;

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du 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