1.7 USE OF A REAL-TIME DECLARATIVE LANGUAGE FOR SYSTOLIC ARRAY DESIGN AND SIMULATION

Nicolas Halbwachs and Daniel Pilaud

INTRODUCTION

The natural tools for describing systolic algorithms consist, on the specification level, of sequences of values defined by difference equations, and on the implementation level, of nets of connected cells. The real-time language LUSTRE, which is under development in Grenoble, combines these two description levels, and seems particularly well-suited for several tasks in the field of systolic array design, namely:
- the initial specification of the algorithm, since a system of difference equations is already (up to some syntactic conventions) a LUSTRE program;
- the description of the final architecture, since the program structuring features of LUSTRE allow an easy description of connected data-flow devices;
- the formalization and proof of the design process, since the mathematical nature of LUSTRE allows a wide range of formal program transformation rules;
- the hardware description of cells, since each cell is a LUSTRE subprogram, which can in turn be viewed as a net of connected operators;
- the simulation of the system at each step of its design, since each description is an executable program.

The basic ideas of LUSTRE are the following:
• Like in LUCID, the well-known non procedural language proposed by Ashcroft and Wadge (1985), each variable in LUSTRE represents an infinite sequence of values, and programs operate globally over sequences.
• LUSTRE is an applicative language: Any program, as well as any operator, is a mapping over sequences. Programs are structured as nets of nodes, which can be viewed as nets of data-flow operators connected with wires. A node declaration specifies a relation between its input parameters and its output parameters, by means of a system of equations. Nodes are instantiated in a functional style, which permits, by simple parameter passing mechanism, the definition of arbitrarily complex nets.
• The language is intended to be synchronously interpreted: each variable possesses its n-th value at the n-th "cycle" of the program. This allows a real-time interpretation of programs, since the execution cycle of a program may be viewed as a physical time unit.
• However, the strict synchronism may be released, by means of two operators: sampling on boolean conditions (clocks), and projection (current value).

In the first section, we shall outline the main features of the language. More detailed presentations may be found in Bergerand et al (1985) or Bergerand (1986). Then, some useful program transformation rules will be presented, which are applied to the traditional examples of convolution product and matrix product, as designed in Mongenet (1985) or Quinton (1983).

MAIN FEATURES OF LUSTRE

As in LUCID, a variable X in LUSTRE may be viewed as an infinite sequence $x_0, x_1, \ldots, x_n, \ldots$ of values, each belonging to a domain D(X) characterized by the type of X. Each domain contains an undefined value, noted nil. A variable X may be defined by means of an equation "X=E", where E is an expression defining a sequence $e_0, \ldots, e_n, \ldots$ of values belonging to D(X), whose interpretation is:

$$\text{for every } n \geq 0,\ x_n = e_n$$

or, from a timed point of view, at any time n, the value of X equals the value of E.

Synchronous Operators

Right hand side expressions may be built by means of variable identifiers, constants (considered as infinite constant sequences) and the operators defined below:

Data Operators: All the operators over domains of values (for instance, boolean or arithmetic operators, if-then-else and case-of-esac conditional operators) are extended to pointwisely operate on sequences: for any operator op, of arity i, the n-th term of the sequence defined by op(X1,X2,...,Xi) is the result of applying op to the n-th terms of X1,X2,...,Xi. All data operators are strict with respect to nil, with the exception of conditional operators, when their operands evaluating to nil do not need to be evaluated.

Synchronous Sequence Operators: If E is an expression, defining the sequence $e_0, \ldots, e_n, \ldots$, then pre(E) defines the sequence nil,$e_0, \ldots, e_{n-1}, \ldots$ If F is an expression of the same type as E, defining the sequence $f_0, \ldots, f_n, \ldots$ then E->F is an expression defining the sequence $e_0, f_1, f_2, \ldots, f_n, \ldots$ For instance, the equation

$$X = 0 \text{ -> } \mathrm{pre}(X) + 1$$

defines the variable whose n-th value x_n satisfies

$$x_n = \begin{cases} 0, & \text{if } n=0 \\ x_{n-1} + 1, & \text{if } n>0 \end{cases}$$

and, so, $x_n = n$.

Program Structuring: Nodes Definition and Instantiation

A node is a subprogram. It receives input variables, computes output variables, and possibly local variables, by means of a system of equations. Node instantiation takes a functional form: if N is the identifier of a node declared with heading

$$\text{node N } (I_1{:}t_1; \ldots; I_p{:}t_p) \text{ returns } (J_1{:}s_1; \ldots; J_q{:}s_q)$$

and if $E_1, \ldots, E_p$ are expressions of types $t_1, \ldots, t_p$, then $N(E_1, \ldots, E_p)$ is an expression of type tuple($s_1, \ldots, s_q$) whose n-th value is the tuple $(j_{1n}, \ldots, j_{qn})$ computed by the node from the input parameters $E_1, \ldots, E_p$. One may write the equation

$$(X_1, \ldots, X_q) = N(E_1, \ldots, E_p)$$

A node which has several output parameters returns a tuple of variables, or a variable of type tuple. LUSTRE is a strongly typed language, but has some polymorphic operators: conditional operators, sequence operators, and the NIL operator. These polymorphic operators may be applied to tuples.

This mechanism for declaring and instantiating nodes allows the definition of whole nets of parallel, synchronous operators. In fact, all the operators may be considered as nodes: For instance, the equation "X = 0->pre(X)+1" describes the net of Fig.1.

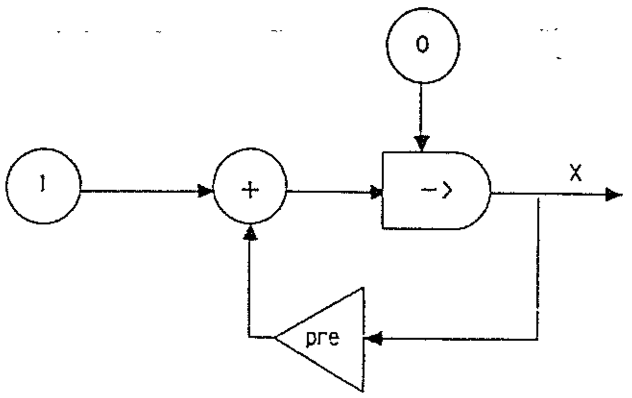


Figure 1

Hence, we adopt the well-known point of view of dataflow nets (Ackerman 1979, Dennis 1974, Kahn 1974) restricted with the synchronous constraint that operators do not communicate through FIFO queues.

Arrays

Of course, for programming systolic algorithms, we need arrays of LUSTRE variables. In LUSTRE, there exist arrays of any dimension, but in order to make possible static check of semantic consistency, their use is restricted by the following constraints:

- Array bounds must be known at compile time;

- Arrays may not be indexed by variables: an index must be an expression made of simple operators (+, −) applied to constants and indexes of "for...let...tel" constructs. Such a construct allows global definition of array elements. For instance, one can write:

PX[0] = X; for i in 1..n let PX[i] = pre(PX[i-1]) tel;

which defines PX[i] to be pre^{i}(X), for i in {0...n}.

Clock Changes

Until now, programs are written in strong connection with their basic execution cycle: all variables in a program evolve with the same frequency. However, there is a strong need of being able to define variables with slower frequency than the basic cycle: for instance, we shall see, in the matrix product example, that the result of the product is available only at some instant. In order to allow such "clock changes", we only need two operators: the sampling according to a boolean condition and the projection.

Sampling: We define a clock to be any boolean variable. If C is a clock and E is an expression, "E when C" is an expression defining the sequence whose n-th term is the value of E at the n-th instant when C is true. "E when C" is said to be on clock C.

Current value: If E is an expression of clock C, then current(E) is an expression whose value at each cycle is the value taken by E at the last cycle when C was true.

The following table (where tt and ff stand for true and false) illustrates these operations:

C =	tt	ff	tt	tt	ff	ff	tt	ff	tt
E =	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
X = E when C =	e_0		e_2	e_3			e_6		e_8
Y = current(X) =	e_0	e_0	e_2	e_3	e_3	e_3	e_6	e_6	e_8

Several remarks must be made about these operations:

- In the above example, X is computed on clock C. This means that the only notion of time that X "knows" is the sequence of cycles during which C is true. As a consequence, the question "what is the value of X when C is not true" makes no sense.
- A consequence of this definition is that two variables may describe the same sequence of values without being equal. Henceforth, a variable will be characterized not only by its sequence of values, but also by its clock. So, with every variable is (syntactically) associated a clock, which may be the always true condition if the variable is renewed at the basic cycle.
- Constants will be considered to be on the basic clock.
- Obviously, the when operator makes it possible that several variables in a program be synchronized according to different clocks. Now, such asynchronous variables may not be used as operands of the same data operator, which operates on terms of the same ranks: In order to operate on variables with different clocks, we shall have first to "project" them onto the same clock, by means of the current operator. The operator current permits operations over asynchronous variables, since if X and X' are variables of respective clocks C and C', then current(X) op current(X') is a legal expression for every data operator op.

SOME TRANSFORMATION RULES

We aim to show that LUSTRE is a suitable formalism both for specifying a problem and for describing a systolic solution. Moreover, the mathematical nature of the language allows the formal derivation of the solution, thanks to some algebraic transformation rules. In the following, x,y,z,... stand for any LUSTRE expression, k stands for any constant expression, and c stands for any boolean expression.

Axioms of sequence operators:

$$x = x \rightarrow x \qquad (1)$$
$$x \rightarrow (y \rightarrow z) = (x \rightarrow y) \rightarrow z = x \rightarrow z \qquad (2)$$
$$\mathrm{pre}(k) = \mathrm{nil} \rightarrow k \qquad (3)$$
$$\mathrm{pre}(x) = \mathrm{nil} \rightarrow \mathrm{pre}(x) \qquad (4)$$

Distributivities: Let op be any n-ary data operator. Then,

$$\mathrm{op}(x_1,\ldots,x_k) \rightarrow \mathrm{op}(y_1,\ldots,y_k) = \mathrm{op}(x_1 \rightarrow y_1,\ldots,x_k \rightarrow y_k) \qquad (5)$$
$$\mathrm{pre}(\mathrm{op}(x_1,\ldots,x_k)) = \mathrm{op}(\mathrm{pre}(x_1),\ldots,\mathrm{pre}(x_k)) \qquad (6)$$
$$\mathrm{op}(x_1,\ldots,x_k) \text{ when } c = \mathrm{op}(x_1 \text{ when } c,\ldots,x_k \text{ when } c) \qquad (7)$$
$$\mathrm{current}(\mathrm{op}(x_1,\ldots,x_k) = \mathrm{op}(\mathrm{current}(x_1),\ldots,\mathrm{current}(x_k)) \qquad (8)$$

Remark: Retiming theorems of Brookes (1984) and Leiserson and Saxe (1981) are strongly related with distributivity of pre and -> with respect to data operators.

From rules (1)(3)(4)(5), we can show the following useful lemma: if op is a strict data operator then

$$\mathrm{op}(\mathrm{pre}(x),k) = \mathrm{op}(\mathrm{pre}(x), \mathrm{pre}(k)) \qquad \text{(Lemma 1)}$$

Booleans and conditional: Boolean calculus is slightly complicated in LUSTRE, because of the third value "nil" that a boolean expression can take. Let us define a total expression to be a expression which never takes the value "nil". We have the following rules:

$$\text{if c then true else false} = \text{c} \tag{9}$$
$$\text{if c then x else y} = \text{if not c then y else x} \tag{10}$$
$$\text{c total} \Rightarrow \text{if c then x else x} = \text{x} \tag{11}$$
$$\text{if true then x else y} = \text{x} \tag{12}$$
$$c_1 \text{ and } c_2 \text{ total} \Rightarrow \text{if c then } c_1 \text{ else } c_2 = (\text{c and } c_1) \text{ or } (\text{not c and } c_2) \tag{13}$$
$$\text{x -> y} = \text{if (true->false) then x else y} \tag{14}$$

Let us apply some of these rules to the design of systolic algorithms, according to the method of Quinton (1983).

CONVOLUTION PRODUCT

The initial equation is

$$y(i) = \sum_{k=0}^{K} x(i-k)\, w(k)$$

which can be translated into a "pseudo-LUSTRE" as

$$y = \sum_{k=0}^{K} \text{pre}^{k}(x)*w(k)$$

Now, the first step of the method proposed by Quinton (1983) consists of rewriting this equation into a system of uniform recurrent equations:

$$Y(i,k) = Y(i,k-1) + W(i,k)*X(i-1,k-1) \quad (k=0\ldots K,\ i>0)$$
$$W(i,k) = w(k) = W(i-1,k) \quad (k=0\ldots K,\ i>0)$$
$$X(i,k) = X(i-1,k-1) \quad (k=0\ldots K,\ i>0)$$
$$Y(i,-1) = 0$$
$$y(i) = Y(i,K)$$
$$X(i,-1) = x(i)$$

This system corresponds to the following LUSTRE equations:

```
for k in [0..K]
let
   Y[k] = Y[k-1] + W[k]*pre(X[k-1]);
   X[k] = pre(X[k-1]);
tel;
Y[-1] = 0 ; y = Y[K] ; X[-1] = x ;
```

This system is not systolic, since Y[k] depends on Y[k-1]. The second step of the design according to Quinton (1983), consists of performing the following variable change:

$$X'(i,k) = X(i-k-1,k) \quad Y'(i,k) = Y(i-k-1,k)$$

which is expressed in LUSTRE by:

$X'[k] = pre^{k+1}(X[k])$; $Y'[k] = pre^{k+1}(Y[k])$;

By substitution in the above program, we get:

$$X'[k] = pre^{k+1}(X[k]) = pre^{k+2}(X[k\text{-}1]) = pre^{2}(pre^{k}(X[k\text{-}1])) = pre^{2}(X'[k\text{-}1])$$

$$Y'[k] = pre^{k+1}(Y[k]) = pre^{k+1}(Y[k\text{-}1] + W[k]*pre(X[k\text{-}1]))$$

$$= pre^{k+1}(Y[k\text{-}1]) + W[k]*pre^{k+2}(X[k\text{-}1]) \quad \text{(from (6) and lemma 1)}$$

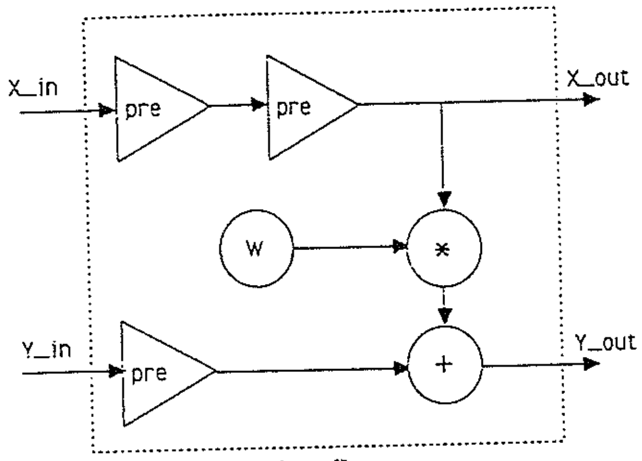
$$= pre(Y'[k\text{-}1]) + W[k]*X'[k]$$

$$X'[-1] = pre^{0}(x) = x$$

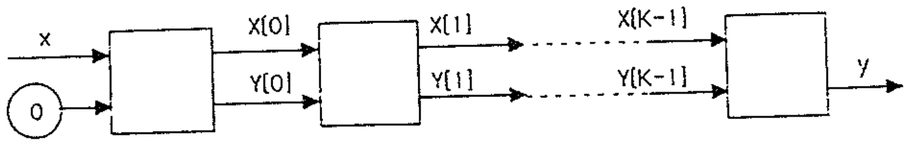
$$Y'[-1] = pre^{0}(0) = 0$$

So, the program becomes

```
for k in 0..K
let
   Y'[k] = pre(Y'[k-1]) + W[k]*X'[k];
   X'[k] = pre(pre(X'[k-1]));
tel;
X'[-1] = x ; Y'[-1] = 0 ; y = Y'[K] ;
```



Basic cell



Overall architecture

Figure 2: Convolution product

Moreover, this program may be structured according to a well-known architecture (Fig. 2): Let us define a node corresponding to the basic cell

```
node CELL (const W: real; Xin, Yin: real) returns (Xout, Yout: real);
let
  Yout = pre(Yin) + W*Xout ;
  Xout = pre(pre(Xin)) ;
tel;
```

then the row of cells is defined by the following main program:

```
node CONVOLUTION (const W: array of real; const K: integer; x: real)
          returns y: real;
var X,Y: array of real;
let
  X[-1] = x;
  Y[-1] = 0;
  y = Y[K];
  for k in 0...K
  let
    (X[k],Y[k]) = CELL(W[k],X[k-1],Y[k-1])
  tel
tel;
```

MATRIX PRODUCT

Given two $n \times n$ matrices A and B, we have to compute the matrix C, such that

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

The system of uniform equations proposed by Quinton (1983) is the following

$$C(i,j,k) = C(i,j,k\text{-}1) + A(i,j,k)^*B(i,j,k)\ ,\ i=1..n\ ,\ j=1..n\ ,\ k=1..n$$

$$A(i,j,k) = A(i,j\text{-}1,k)\ ,\ i=1..n\ ,\ j=1..n\ ,\ k=1..n$$

$$B(i,j,k) = B(i\text{-}1,j,k)\ ,\ i=1..n\ ,\ j=1..n\ ,\ k=1..n$$

with

$$C(i,j,0) = 0\ ,\ A(i,0,k) = a_{ik}\ ,\ B(0,j,k) = b_{kj}\ ,\ i=1..n\ ,\ j=1..n\ ,\ k=1..n$$

First program: Let us consider k as the time index. Let a[i] represent the sequence (a_{ik}, k=1..n) and b[j] represent the sequence (b_{kj}, k=1..n). The above system provides the program:

```
for i in 1..n, j in 1..n
let
  C[i,j] = 0 -> pre(C[i,j]) + A[i,j]*B[i,j];
  A[i,j] = A[i,j-1] ; B[i,j] = B[i-1,j];
tel;
for m in 1..n
let A[m,0] = pre(a[m]); B[0,m] = pre(b[m]) tel;
```

Moreover, we have to compute the instant when the value c_{ij} is available. Let e[i,j] be n-t at every instant t. So c_{ij} is available when e[i,j]=0. These variables are defined by:

```
for i in 1..n , j in 1..n let e[i,j] = n -> pre(e[i,j] - 1) tel;
```

and the product A×B is the matrix c defined by

```
for i in 1..n , j in 1..n let c[i,j] = C[i,j] when (e[i,j]=0) tel;
```

Time change: In order to get a systolic program, let us make the following time change: t'= i+j+t-1. This means that any array element X[i,j], that was previously computed at time t, will now be computed at time i+j+t-1. So, any array element X[i,j] has to be replaced by X'[i,j] where

$$X'[i,j] = \mathrm{pre}^{i+j-1}(X[i,j])$$

Let init = true -> false. Using rules of section 2, we get, for i=1..n and j=1..n:

$$C'[i,j] = \text{if } \mathrm{pre}^{i+j-1}(\mathrm{init}) \text{ then } 0$$
$$\text{else } \mathrm{pre}^{i+j}(C[i,j]) + \mathrm{pre}^{i+j-1}(A[i,j]) * \mathrm{pre}^{i+j-1}(B[i,j])$$
$$A'[i,j] = \mathrm{pre}^{i+j-1}(A[i,j-1])$$
$$B'[i,j] = \mathrm{pre}^{i+j-1}(B[i-1,j])$$
$$e'[i,j] = \text{if } \mathrm{pre}^{i+j-1}(\mathrm{init}) \text{ then } n \text{ else } \mathrm{pre}^{i+j}(e'[i,j]) - 1$$

and for m = 1..n,

$$A'[m,0] = \mathrm{pre}^{m}(a[m])$$
$$B'[0,m] = \mathrm{pre}^{m}(b[m])$$

Let r(i,j) = pre^{i+j-1}(init). We have r(1,0) = init, r(1,j) = pre(r(1,j-1) and r(i,j) = pre(r(i-1,j). By substitution, the program becomes:

```
for i in 1..n , j in 1..n
let
   C'[i,j] = if r[i,j] then 0 else pre(C'[i,j]) + A'[i,j]*B'[i,j];
   A'[i,j] = pre(A'[i,j-1]); B'[i,j] = pre(B'[i-1,j]);
   e'[i,j] = if r[i,j] then n else pre(e'[i,j]) - 1;
tel;
for i in 2..n , j in 1..n let r[i,j] = pre(r[i-1,j]) tel;
r[1,0] = init;
```

assuming that, for m in 1..n, A[m,0] = pre^{m}(a[m]) and B[0,m] = pre^{m}(b[m]). These inputs are defined by means of two triangular arrays aa and bb, such that aa[i,j] = pre^{j}(a[i]) and bb[i,j] = pre^{i}(b[j]):

```
for p in 1..n
let
  aa[p,0] = a[p] ; bb[0,p] = b[p];
  for q in 1..p
  let aa[p,q] = pre(aa[p,q-1]) ; bb[q,p] = pre(bb[q-1,p]) tel;
  A[p,0] = aa[p,p] ; B[0,p] = bb[p,p];
tel;
```

The new output c' is defined by

```
for i in 1..n , j in 1..n
let c'[i,j] = C'[i,j] when (e'[i,j]=0) tel
```

The final architecture can now be described. The basic cell is the following node:

```
node CELL ( A_in, B_in: real; R_in: bool)
        returns (A_out, B_out, C_out: real; R_out: bool);
var E: bool; C: real;
let
  A_out = pre(A_in); B_out = pre(B_in);
  C_out = C when (E=0); R_out = pre(R_in);
  C = if R_out then 0 else pre(C) + A_out*B_out;
  E = if R_out then n else pre(E) - 1;
tel;
```

and the main program may be written:

```
for i in 2..n , j in 1..n
let
  (A[i,j], B[i,j], C[i,j], R[i,j]) = CELL(A[i,j-1], B[i-1,j], R[i-1,j]);
tel;
for j in 2..n
let
  (A[1,j], B[1,j], C[1,j], R[1,j]) = CELL(A[1,j-1], B[0,j], R[1,j-1]);
tel;
(A[1,1], B[1,1], C[1,1], R[1,1])
        = CELL(A[1,0], B[0,1], init);
for p in 1..n
let
  A[p,0] = aa[p,p] ; B[0,p] = b[p] ;
  aa[p,0] = a[p] ; bb[0,p] = b[p];
  for q in 1..p
  let aa[p,q] = pre(aa[p,q-1]) ; bb[q,p] = pre(bb[q-1,p]) tel;
tel;
```

Remark: In the above program, the variable init is assumed to be "true -> false". In fact, it may be true whenever C[1,1] has been computed, then making the program re-entrant and pipe-line.

CONCLUSION

Of course, we do not pretend that the use of LUSTRE will help in finding good systolic algorithms. However, expressing such an algorithm in a programming language will allow the simulation of the algorithm at each step of its design, and force the designer to make precise some details, which are often left in the dark in the literature. Examples of such details are the following:

- when and where must the inputs be provided?

- when and where are the outputs available?

- what is the precise meaning of the absence of a value on some wire? In usual descriptions, such an absent value is interpreted sometimes as a neutral element, sometimes as a zero element of any operator.

A prototype compiler of LUSTRE has been written, but, as the programming of systolic arrays is not our main application field, this first version does not allow arrays. A new version will soon follow which will contain arrays, with the restriction that their bounds must be known at compile-time.

Other works around LUSTRE are either in progress or strongly considered:

- The design of a graphic interface, allowing the visualization of the execution of a program on the net of nodes. It will be a good demonstration tool for systolic algorithms.

- The code generation for parallel machines.

- A system for mechanical help in formal program transformations.

- Circuit design from LUSTRE programs: of course, the dataflow net is a good basis for such a design, but other implementations can be derived by considering a program from a finer time scale, thus allowing multiple uses of some operators in the same cycle.

REFERENCES

Ackerman W.B. 1979:"Data flow languages". Proc. AFIPS Conf., Arlington.

Ashcroft E.A., Wadge W.W. 1985: "LUCID, the data-flow programming language", Academic Press.

Bergerand J.L., Caspi P., Halbwachs N., Pilaud D., Pilaud E. 1985: "Outline of a data-flow programming language", Proc. Real-Time Systems Symp., San Diego (Ca.).

Bergerand J.L. 1986: "Lustre: un langage déclaratif pour le temps réel", Thesis, Institut National Polytechnique de Grenoble.

Brookes S.D. 1984: "Reasoning about synchronous systems" R.R. CMU-CS-84-145, Dept. of Computer Sci., Carnegie-Mellon Univ.

Dennis J. 1974: "First version of a data flow procedure language". Proc. Colloque sur la programmation, LNCS nr. 19.

Kahn G. 1974: "The semantics of a simple language for parallel processing". Proc. IFIP Congress.

Leiserson C.E., Saxe J. 1981: "Optimizing synchronous systems", Foundations of Computer Science.

Mongenet C. 1985:"Une méthode de conception d'algotithmes systoliques. Résultats théoriques et réalisation", Thesis, Institut National Polytechnique de Lorraine.

Quinton P. 1983:"The systematic design of systolic arrays", R.R.193, IRISA, Rennes.