SIGNAL : a data flow oriented language for signal processing

Albert Benveniste, Patricia Bournai, Thierry Gautier, Paul Le Guernic

INRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tel : (3) 954 90 20

Rapports de Recherche

N° 378

SIGNAL: A DATA FLOW ORIENTED LANGUAGE FOR SIGNAL PROCESSING

Albert BENVENISTE
Patricia BOURNAI
Thierry GAUTIER
Paul LE GUERNIC

Mars 1985

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Publication Interne n° 246

Janvier 1985
62 pages

SIGNAL : A DATA FLOW ORIENTED LANGUAGE FOR SIGNAL PROCESSING

Paul LE GUERNIC, Albert BENVENISTE, Patricia BOURNAI
Thierry GAUTIER

IRISA/INRIA
Campus de Beaulieu
35042 RENNES CEDEX

ABSTRACT : We present the language SIGNAL, which is a data-flow oriented real time, synchronous, side effect-free language suited to the expression and recovery of the parallelism in signal or image processing algorithms. The language is intended to be at the same time an executable simulation language, and a specification of a virtual machine implementing the algorithm. The language is semantically sound, and is suitable to perform program transforms, a major requirement when the ultimate goal is an aid to the architecture design.

RESUME : Nous présentons le language SIGNAL qui est un langage temps réel, synchrone, orienté flot de données, adapté à l'expression et à l'exploitation du parallélisme dans les algorithmes de traitement du signal et de traitement d'image. Le but du langage est d'être à la fois un langage de simulation et une spécification de la machine virtuelle sur laquelle sera implanté l'algorithme simulé. Le langage est défini formellement et permet des transformations de programmes, ce qui est indispensable lorsque l'objectif final est une aide à la conception d'architecture.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

SIGNAL : A DATA FLOW ORIENTED LANGUAGE FOR SIGNAL PROCESSING

Paul LE GUERNIC, Albert BENVENISTE
Patricia BOURNAI, Thierry GAUTIER

Publication n° 246 - Janvier 1985

1. INTRODUCTION.

The need for a fast implementation of signal or image processing algorithms is growing up very rapidly. The main concerned areas are: real-time control of industrial processes, telecommunications, videocommunications, speech and image coding, speech processing and real-time computer vision, biomedical applications, and mostly signal and image processing encountered in radar and sonar systems.

On the other hand, the availability of

- new specialized microprocessors, such as the audio-bandwidth range signal processing oriented microprocessors,
- faster technologies such as VHSIC,
- new silicon compilers

is increasing even more rapidly.

But a too long delay still remains for the fielding of these new tools and processors when the signal processing task definition and specification is considered as the entry point of the chain. This delay is mostly due to the relative lack of CAD tools from signal processing task specification to architecture specification.

Hence, there is a need for a coherent set of tools with

- as input: formal specifications of signal processing tasks,
- as output: formal descriptions of multiprocessor architectures, or special purpose architectures (such as systolic ones),

and further providing a (partial) validation of the correctness of the implementation with respect to the specification. For such a purpose, a basic tool is a suitable language for signal processing task specification and execution: the

purpose of the present paper is to present such a language, the language SIGNAL.

Complex signal processing tasks, such as encountered in speech and image processing, typically involve

1/ the transform of signals by highly regular elementary algorithms (filters, LPC, FFT,...)

2/ some decision rules based on the monitoring of the various signals (tests,...), thus eventually resulting in the running of signals with different clocks.

Consequently, our main objectives in designing our language were the following:

* SIGNAL should be able to describe any real-time signal processing task, involving possibly the running of signals with different clocks;
* SIGNAL should allow the programmer to specify, and the CAD system to recognize in an easy way the parallelism present in a given task.

By the way, the SIGNAL specification of a task should be considered as the description of a virtual machine implementing this task. Finally, SIGNAL should be more than a specification language, namely an executable language: this would allow the programmer to use SIGNAL as a simulation language for the validation of the algorithms at the highest level.

The paper is organized as follows. In the first section, the basic principles supporting the language are presented. The second section is devoted to the presentation of the tools for constructing block-diagrams, i.e. static networks on which the various signals flow. In the third section, the synchronization mechanisms of SIGNAL (i.e. its temporal aspects) will be presented. In the fourth section, we shall discuss the recovery of the dependence graph. Finally, the fifth section will be devoted to the presentation of some examples.

1.1. Basic principles supporting the language SIGNAL

Recall the main objectives we have in mind in designing the language SIGNAL.

(i) SIGNAL should be able to describe any real-time task (relevant to signal processing), that is to say, according to Young (1982), "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specifiable delay". By "specifiable", we have in mind that it is possible to know in advance this delay prior to the running of the task.

(ii) the timing mechanisms should be described in an entirely synchronous way. The major reason for this requirement is that our primary interest lies in the specification of algorithms independently of their eventual implementation, so that we can consider that every action is instantaneous, i.e. has a zero duration. As a matter of fact, this is the most easy way to prevent from any non-determinism in the language, due to the unknown duration of the actions (as it has been pointed out by Berry & al.(1983),(1984)).

(iii) The specification, or execution, of a task should not require any prior knowledge of a universal time reference. The time has rather to be nothing but the ordering of the input stimuli or data of the given task. Hence, the modularity of the language (a highly desirable feature) requires that the notion of time be multiform, since the cooperation between several subtasks (with their own local timing) will result in the modification of their time references, as it will be shown later.

(iv) SIGNAL should allow the programmer to specify in an easy way the parallelism present in a given task. In other words, the expression of the task

in SIGNAL should reflect as much as possible the dependencies between the data and operations. This feature will be of primary interest for the recovery of parallelism and pipelining capabilities.

Finally, as any modern language, SIGNAL should provide all the state of the art characteristics to ensure modularity, strong typing, readability, security. Moreover, the temporal instructions of SIGNAL should be built from a small set of primitive statements, for which a formal semantics must be given.

Some existing languages (Young(1982);ESTEREL, Berry & al.(1983)) almost satisfy these conditions. Related to the classical *real-time languages* , these languages concentrate on a microscopic and explicit description of the timing of the actions. As a consequence, the recovery of the data dependencies becomes difficult for complex tasks (point (iv)).

On the other hand, *applicative languages* are tailored to express the parallelism existing in a task (LISP, Mc Carthy (1966); LUCID, Ashcroft (1977); FP, Backus (1978); DSM, Cremers & Hibbard (1982)). But these languages are not well suited to the timing problems.

Related to the family of the applicative languages is the family of the *data flow languages* (Dennis (1974); Kahn (1974); VAL, Mc Graw (1982); Ackerman (1979); CAJOLE, Hankin (1981)). In the data flow languages, the instruction firing rule is simply based on the availability of data. These languages are suited to the processing of infinite files of data. They do not require any explicit specification of the timing of each action, thanks to this automatic firing rule. However, pure data flow languages are asynchronous, so that they do not fulfill the requirement (ii).

The language SIGNAL exhibits several of the interesting features of both the real-time and data flow languages. These are obtained in the following way.

A SIGNAL program is

(a) *the hierarchical specification of a process, i.e. of a static oriented network* (or block-diagram in the framework of signal processing) *expressing exactly the data dependencies*. Data flow along the paths of the network; at the nodes, actions are performed by (sub)processes, or black-boxes in the signal processing framework.

(b) *the description of the elementary processes, referred to as generators, acting at each node*. Among the set of all the generators, we shall distinguish the *temporal generators* , which are specific to SIGNAL. The purpose of these temporal generators is to express the relative timing of the various signal flows in an entirely synchronous way.Thus the availability of a signal is completely characterized by a clock, so that we shall be able to refer precisely to the time at which a given action is performed.

(c) *among the set of the actions wich are performed at the same instant, the firing rule is determined according to the principles of data flow* (i.e. automatic firing through the availability of data at the input ports). Thanks to this last principle, the timing is completely specified by the static relationships between the various clocks involved in a task; the internal timing of a clock (i.e. the detailed specification of the events subject to a given clock) can be completely implicit, contrary to the classical real time languages.

To summarize, SIGNAL is a real time, synchronous, side effect free language suited to the expression and recovery of parallelism.

2. CONSTRUCTION OF THE STATIC NETWORKS

In order to improve the understanding, the reader who is familiar with signal processing or automatic control can refer to the building of block diagrams through the interconnection of black boxes. The formal semantics of this part of the language is given in Le Guernic (1982).

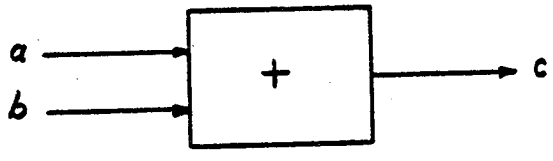
2.1. Processes and generators.

2.1.1. Generators.

A generator is specified by

- its name
- a list of named input ports
- a list of named output ports

fig. 2.1: graphic description of a generator



Usually, generators will be completely specified by SIGNAL expressions. For example c := a+b will refer to the generator depicted in the figure 2.1. The naming of the input ports is free, but the naming of the output ports is subject to the following constraint

(C.1) two different output ports must have different names.

The motivation for introducing the condition (C.1) will be enlightened later. The

generators will be the elementary prototype processes.

2.1.2. Processes.

A process is a network of interconnected generators. These connections are subject to the following constraint

(C.2) an input port can be connected to at most one output port.

This condition is intended to prevent from an unknown shuffling of the data at the considered intput port, a situation which would result in nondeterminism in the language. On the other hand, an output port can be connected to one or several input ports; in the later case, the same data will be broadcasted to these input ports.

A process is characterized by

- a name P (optional)
- a list of named input ports, denoted by, say, ?a,b,c,...
- a list of named output ports satisfying (C.1), denoted by, say, !x,y,z,...
- a body describing how the network was constructed.

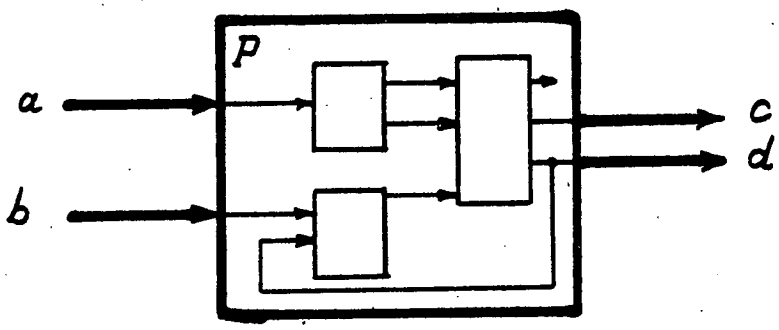
The symbols ? and ! , together with the corresponding list of named input and output ports, define the *interfaces* of the process. These interfaces are subject to the following constraints. No constraint is imposed for the naming of the output ports, but

(C.3) a nonnamed input port must be connected to some internal output port of the process; conversely, a named input port cannot be connected to any internal output port of the process.

The first part of the condition is intended to prevent from a possible starvation; the second part of the condition prevents from a possible further connection of this input port to some new output port (which would result inn a shuffling of the

data at this input port, a situation we want to avoid). The interfaces (i.e. the set of named input and output ports) will be used to interconnect processes, thus resulting in a hierarchical construction of new processes. The figure (2.2) shows the interfaces of the process P, and gives a graphic description of its body.

fig. (2.2): block-diagram of the process P{?a,b !c,d}, P= 'body'



2.1.3. Connecting processes: the basic principles.

These principles are the following:

- *an existing connection cannot be removed;*
- *connections of processes are obtained through the identity of the names of their corresponding output and input ports.*

The interconnections of ports of several processes result in a new process. This is exactly the way to build hierarchically new processes from generators or already defined processes. The next paragraphs will be devoted to the description of operators for connecting processes, we shall refer to as *interconnection operators*.

In the sequel, we sha. use the following notation. Given a process

P { ? list ! list }

we shall denote respectively by

? (P), or simply ? P (! (P) or simply ! P)

the set of the input (resp. output) ports of P.

2.2. Primitive operators.

These operators are the only operators for which a formal semantics has been given in Le Guernic (1982) and in Gautier (1984). We shall here present these operators in an informal way. In order to know the result of an operator, it is sufficient to know the interfaces of the involved processes. There are three primitive operators.

2.2.1. Binary operator PARALLEL.

Given two processes P and Q, the operator PARALLEL replaces any pair of input ports of P and Q with the same name by a single one, thus broadcasting the same values on these two different ports. The result is a new process if and only if it satisfies the condition (C.1).

Notation: given P and Q,

$R = P \,\&\, Q$

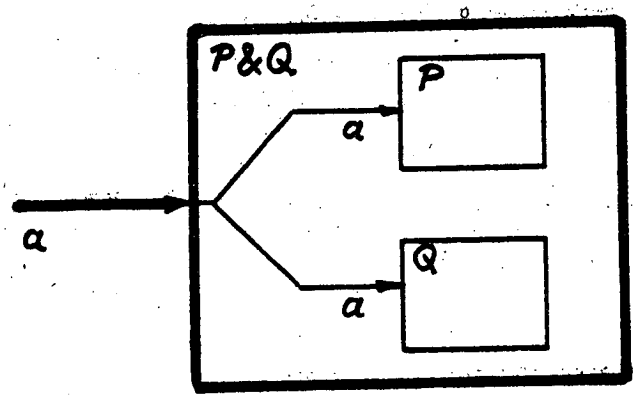
is defined if and only if

$! P \cap ! Q = \phi$.

In this case,

$? R = ? P \cup ? Q$, and $! R = ! P \cup ! Q$

fig. (2.3): operator PARALLEL, graphical example



properties: PARALLEL is commutative and associative.

2.2.2. 1-ary operator LOOP.

Given a process P and a name x belonging to both sets of named input and output ports of P, the operator LOOP connects the output port named x to all the input port(s) named x. The result is a new process. In any other case, LOOP has no effect.

Notation: given

P { ? x, list1 ! x, list2 } ,

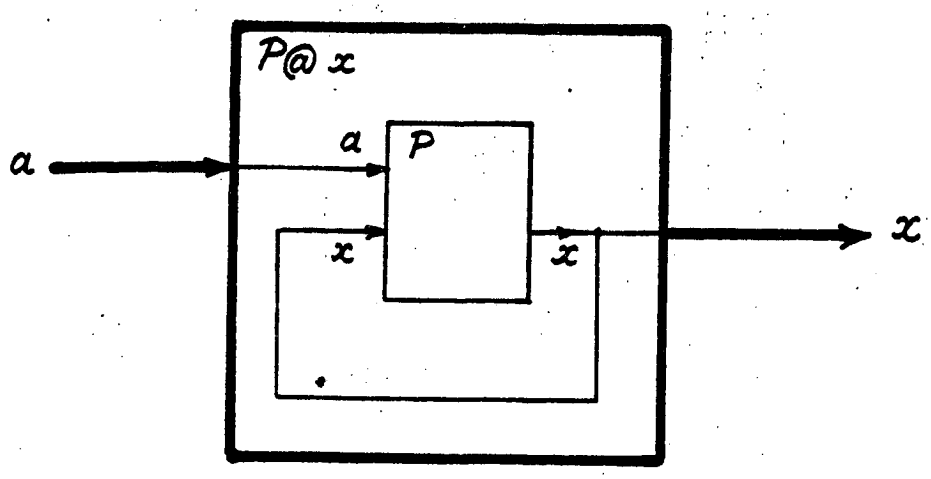
where x is neither a member of list1 nor of list2, then

Q = P ⊛ x

results in

Q { ? list1 ! x, list2 }

fig.(2.4): operator LOOP, graphical example



2.2.3. 1-ary operator RELABELLING.

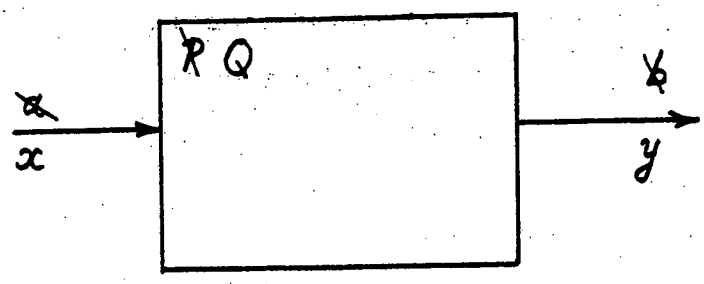
The operator RELABELLING modifies the names of the ports of a process. The purpose of this operator is to prepare future connections, or to prevent from future connections. The relabelling of a port by no name results in the masking of this port, if this port is an output port, and has no effect in the case of an input port (because of the condition (C.3)).

Notation: given a process P,

Q = P ? a : b (resp. Q = P ! x : y)

results in the substitution of b to a (resp. y to x) in the list of the names of the input ports (resp. output ports) of P.

fig.(2.5): operator RELABELLING, graphical example



2.2.4. Priority rules between operators.

• The 1-ary operators have priority over the binary operators.

• There is no need for parentheses when several 1-ary operators are successively used: the priority is indicated by the positions.

• There is no priority between different binary operators: parentheses must be used .

These are the only primitive operators of this part of the language SIGNAL. The next paragraphs are devoted to the presentation of derived operators. All these operators will be defined as macros from the primitive ones.

2.3. Derived operators.

They will generally be presented according to the following scheme: informal presentation ,notation, formal definition as a macro, graphical example.

2.3.1. Operators derived from RELABELLING.

• Notation and definition:

P ! L (resp. P ? L)

where L is a list of notations $a_i:b_i$ results in a simultaneous substitution of $!b_i$ (resp. $?b_i$) to $!a_i$ (resp. $?a_i$) in the interfaces of P. For example

P ? a : b , b : c = P ? a : x ? b : c ? x : b

which is different from P ? a : b ? b : c .

P / L

where L is a list $a_1, \ldots, a_n$ of names result in the masking of all the labels $!a_i$ in P; the input names are not changed because of the condition (C.3).

P : L

where L is a list $a_1,\ldots,a_n$ of names result in the masking of all the output labels of P, except the mentioned ones.

2.3.2. Operator derived from LOOP.

P @ L ,

where L is a list $a_1,\ldots,a_n$ of names is equivalent to

P @ a_1...@a_n .

2.3.3. Binary operator COMPOSITION.

Given two processes P and Q, the COMPOSITION connects the output ports of P (resp.Q) to the input ports of Q (resp. P) with the same name. If the result satisfies (C.1), a new process is obtained.

* Notation:

R = P | Q

* Definition as a macro: let us consider

$P\{?\ x_1,\ldots,x_p,list_{?P}\ !\ y_1,\ldots,y_q,list_{!P}\}$

and

$Q\{?\ y_1,\ldots,y_q,list_{?Q}\ !\ x_1,\ldots,x_p,list_{!Q}\}$

where

$list_{?P} \cap list_{!Q} = \phi$

$list_{?Q} \cap list_{!P} = \phi$

$list_{!P} \cap list_{!Q} = \phi$.

Choose

$y'_1,\ldots,y'_q$

such that

$y'_1,\ldots,y'_q \cap x_1,\ldots,x_p = \phi$.

Then

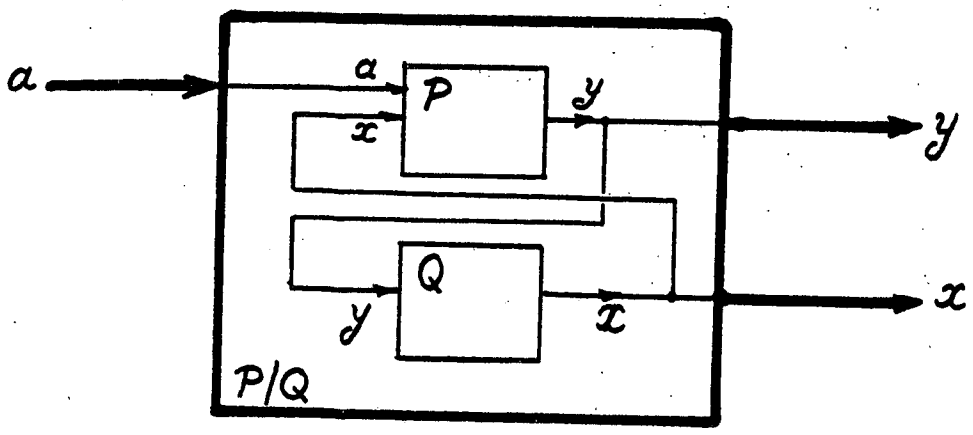
$P \mid Q =$

$(P\ !\ y_1:y'_1,\ldots,y_q:y'_q\ \&\ Q\ ?\ y_1 : y'_1,\ldots,y_q : y'_q)$

$\circledast\ y'_1,\ldots,y'_q,x_1,\ldots,x_p$

$!\ y'_1 : y_1,\ldots,y'_q : y_q$

fig.(2.6): operator COMPOSITION, graphical example



* Properties: COMPOSITION is commutative and associative.

2.3.4. Binary operator SEQUENCE.

Given two processes P1 and P2, their SEQUENCE is always defined, and is obtained as follows: every named output port of P1 is connected to the input port(s) of P2 with the same name; then, the condition (C.1) is satisfied through the automatic masking of every named output port of P1 with a name occurring in the set of the named output ports of P2.

* Notation:

Q = P1 ; P2

* Formal definition as a macro: given

P1 {? $list_{?P1}$! $x_1,\ldots,x_p,y_1,\ldots,y_q,z_1,\ldots,z_r,list_{!P1}$ } and

P2 {? $y_1,\ldots,y_q,z_1,\ldots,z_r,list_{?P2}$! $x_1,\ldots,x_p,z_1,\ldots,z_r,list_{!P2}$ } ,

where different symbols refer to different names, and

$list_{!P1} \cap list_{!P2} = \phi$

choose

$x'_1,\ldots,x'_p$, $z'_1,\ldots,z'_r$

such that $x'_1,\ldots,x'_p \cap x_1,\ldots,x_p = \phi$

$z'_1,\ldots,z'_r \cap z_1,\ldots,z_r = \phi$

then

P1 ; P2 ≡

(P1 ! $x_1{:}x'_1,\ldots,x_p{:}x'_p, z_1{:}z'_1, \ldots, z_r{:}z'_r$

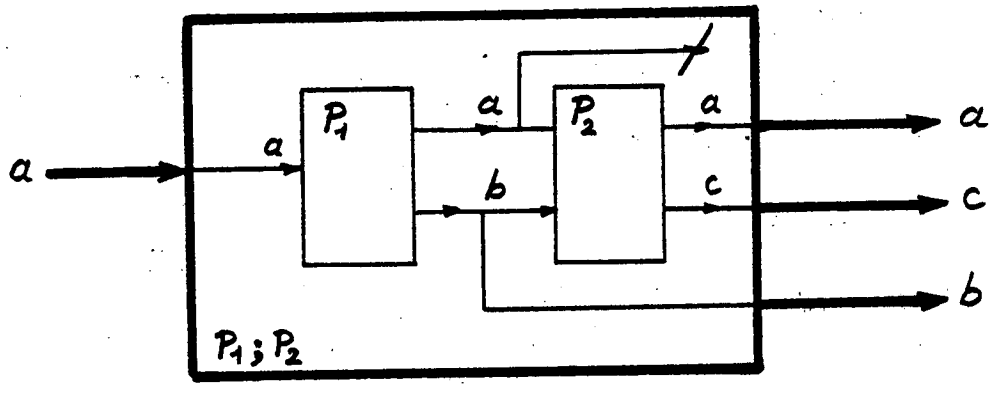
&

P2 ? $z_1{:}z'_1,\ldots,z_r{:}z'_r$)

@ $y_1,\ldots,y_q, z'_1,\ldots,z'_r$

: $x_1,\ldots,x_p, y_1,\ldots,y_q, z_1,\ldots,z_r, list_{!P2}$

fig. (2.7): operator SEQUENCE, graphical example



* Properties: SEQUENCE is associative, but not commutative. It is always defined. Inside a sequence, names of ports can be used almost like variables in classical sequential languages. For example,

a := b←c ; c := b←a

can be interpreted as the corresponding sequence of Pascal instructions; the only difference is that the values carried by the input port c are saved.

2.3.5. Regular arrays of processes.

We give here facilities for constructing regular arrays of processes indexed by a parameter.

* Notation: the generic syntax is the following

do until n **of** P(i) **od**

where

...= **par** for PARALLEL

seq for SEQUENCE

com for COMPOSITION

n is a parameter of integer type, $1 \leq i \leq n$ is an integer, and P(i) is an indexed process with parameter i.

* Formal definitions as macros.

dopar i **until** n **of** P(i) **od** ≡ P(1) **&**...**&** P(n)

doseq i **until** n **of** P(i) **od** ≡ P(1) ;...; P(n)

docom i **until** n **of** P(i) ≡ P(1) |...| P(n)

* Graphical examples: let

P { ? x,y ! x,y }

P = "body"

be given.

1/ Then

(**doseq** i **until** n **of**

(**doseq** j **until** m **of** P ? y:y[j] ! y:y[j] **od)**

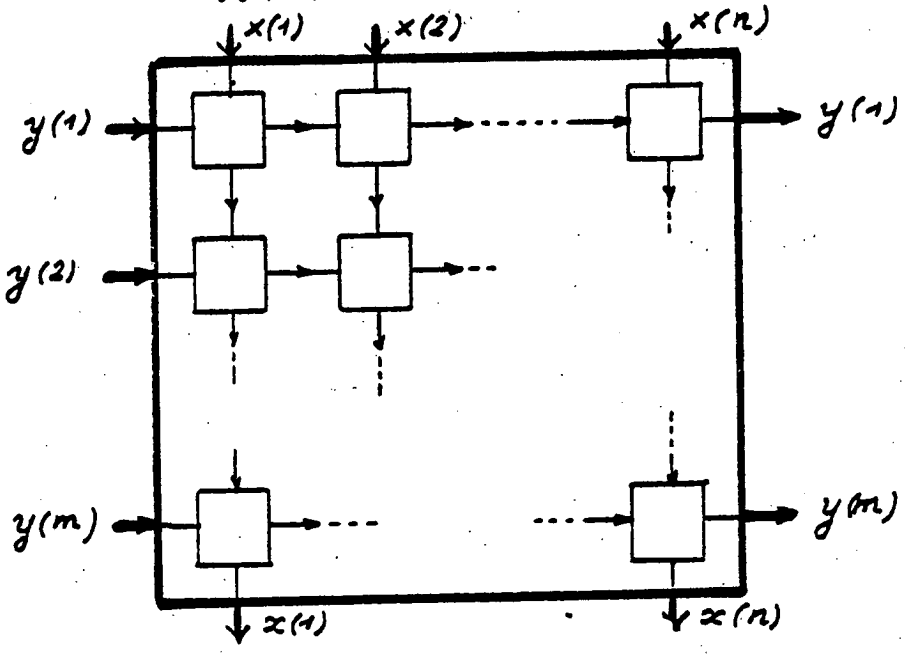
? x:x[i] ! x:x[i]

od)

: x[i],...x[n],y[i],...y[m]

is the following rectangular array of copies of the process P:

fig. (2.8): rectangular array with DOSEQ

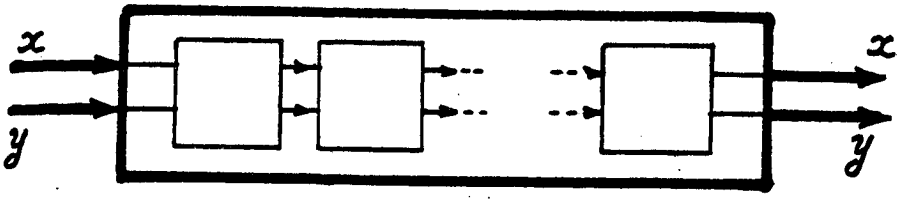


2/ second example:

doseq i **to** n **of** P **od**

is the linear array:

fig.(2.9): linear array with DOSEQ



3/ Third example:

(**docom** i **to** n **of**

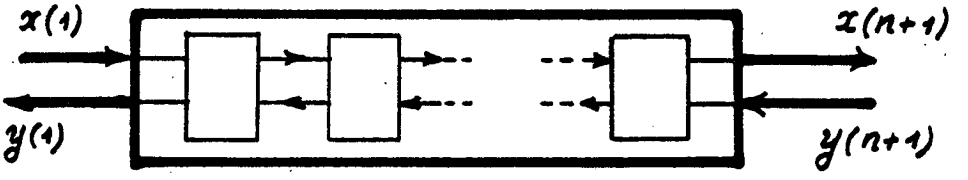
P ? x:x[i],y:y[i+1]! x:x[i+1],y:y[i]

od)

:y[1],x[n+1]

is the cascade:

fig.(2.10): linear array using DOCOM



The following table summarizes the interconnection operators.

CONNECTION OPERATORS		
name	syntax	effect
LOOP	P@x P@List	see fig (2.4)
RELABELLING	P ?a:b !x:y P/a P:a	see fig (2.5)
PARALLEL	P&Q	see fig (2.3)
COMPOSITION	P|Q	see fig (2.6)
SEQUENCE	P1;P2	see fig (2.7)
REGULAR ARRAYS	dopar i until n of P(i) doseq i until n of P(i) docom i until n of P(i)	P(1)&...&P(n) P(1);...;P(n) P(1)|...|P(n)

Recall that the primitive operators are LOOP, RELABELLING, PARALLEL.

These operators are sufficient for constructing any static network. We have described the first part of the language SIGNAL. Note that these static networks are entirely abstract, as it is clear in the corresponding formal model presented in Le Guernic (1983)and Gautier (1984). As a consequence, this part of the language could be used for the description of static networks of completely different nature than the present ones. For example, it could be used for building continuous time signal flow graphs, or block-diagrams for the specification of plants (see Elmqvist (1978)), provided that the corresponding static network be oriented; this is however not the case when the nodes realize equilibriums described by implicit equations of the form F(x,y)=0, as in Elmqvist (1978).

3. THE TEMPORAL GENERATORS, AND THE CORRESPONDING INSTRUCTION SET OF SIGNAL.

This is the most important and original part of the language SIGNAL. Among the whole set of generators of SIGNAL, we shall distinguish two subclasses:

* the subclass of *functions* : this subclass contains the elementary instantaneous tranformations on the data (+, *, and, or,...). Functions are not specific to SIGNAL, but are chosen for a specific version of the language, according to the target application. For example, the software simulation of signal processing algorithms could require the classical FORTRAN- or PASCAL-like arithmetic and boolean operators. But, for the study of the rounding effects, it can be desirable to provide the programmer with "mult" or "add" with a specifiable precision.

* the subclass of *temporal generators* : temporal generators are specific to SIGNAL, whatever the version of the language is concerned. These generators are the basic tools for the description of the synchronization mechanisms between the various signals. The present chapter will emphazise on the presentation of the temporal generators of SIGNAL.

3.1. SIGNALS AND CLOCKS: THE CLOCK ALGEBRA.

3.1.1. Informal description of the synchronization mechanisms.

They are based on the following notions:

* *flow* : a flow is an ordered file of typed data of unspecified length. Given a process (i.e. a static network built according to the preceding chapter), flows run along the paths of this process, and are transformed at the

generators according to the principles of synchronism we shall now describe.

* *clock* : to every flow is associated a clock, i.e. a *relative* specification of the instants at which successive values of the flow are available; values are available at these instants only, i.e. are *not persistent* , unlike variables in classical languages. The specification of the clocks is *relative* in the sense that clocks will only be constrained to satisfy some relationships defined in a static way. These relationships between clocks will be a byproduct of the interconnection of temporal generators.

* *signal* : roughly speaking, a signal is a pair (flow, clock), where the clock specifies the instants at which the data are available.

The reason for avoiding any absolute definition of clocks (i.e. any reference to some "universal" clock) is the following. It is desirable and natural to consider that every process has its own local time reference (usually the ordering of its input stimuli). But the interconnection of processes can result in a modification of this time reference, through an embedding in another timing environment. As a consequence, the use of any universal clock is a major drawback when modularity is desired for the language. Hence, we join the approach developped in Berry (1984). As a difference, we require the relationships between the various clocks to be specified in a static way. A formal definition of clocks will be presented . We shall now introduce the calculus we need for specifying the timing mechanisms: the *clock algebra*.

3.1.2. The clock algebra.

We shall first introduce informally and motivate this calculus. Then, we

shall summarize the objects and axioms of this algebra.

* **The partial order on the set of the clocks:** given two signals x and y, we shall say that x is dominated by y, and denote it by

$$x \leq y$$

if the set of the instants at which x is available is contained in the set of the instants at which y is available (i.e. x is more seldom than y) *whatever the values carried by x and y are*. Hence, the relationship $x \leq y$ can be checked in a static way, before the running of the program. Note that this order is different than the order used by Caspi & Halbwachs (1982), the later one referring to the relative precedence of events, and being intended to the study of the causality.

If $x \leq y$ and $y \leq x$, we shall say that x and y are *simultaneous* and we shall denote this property by

$$x = y$$

Clearly, "=" is an equivalence relation, and we shall refer to **clocks** to distinguish the corresponding equivalence classes. The set of the signals, endowed with the partial order $\leq$ *is generally not a lattice* as we shall see later, for the following reason: some processes accept signals for which the relative timing of their input signals is not specified. Such processes are said to be *non synchronized* and cannot be ran as main programs, while the others are said to be *synchronized*, and can be executed as main programs.

We shall say that two signals x and y are *compatible*, denoted by

$$x \sim y$$

if there exists another signal a which dominates both x and y. Note that "~" is not an equivalence relation, for the transitivity being not satisfied. On the other hand, by Zorn's lemma, the set of the signals dominated by a single

one is a lattice with respect to the partial order "≤" we have defined. In this case, it will be possible to introduce the usual operations, the "sup" and "inf", respectively denoted by

$\cup, \cap$

Note that, since there is no universal maximal element, there is no reasonable definition for "not", a difficulty we shall further investigate later.

* **The set of pure clocks:** among the set of the signals, we shall distinguish the *pure clocks* , which are signals with a constant dummy value (which has no importance). Obviously, pure clocks are typical candidates to represent a clock, i.e. an equivalence class of simultaneous signals.

This simple algebra will be sufficient for deriving all the possible clocks dominated by a single one. Further developpments will be necessary to allow *the oversampling* of signals. This will be achieved by introducing the *multiplexing* of signals. However, the underlying timing mechanisms are difficult to properly define, and this notion is further under investigation. Its presentation will be the subject of a subsequent paper.

3.2. THE PRIMITIVE TEMPORAL GENERATORS OF SIGNAL.

We shall here present the primitive temporal generators of SIGNAL. Some of these primitive generators will not be provided to the programmer, more userfriend macros will be provided instead. Hence, the syntax we shall introduce for denoting these generators has to be considered as a formal calculus rather than as an instruction set. Following Berry & Cosserat (1984), a formal semantics of these generators and of the synchronization mechanisms wll be presented in a subsequent paper.

The primitive generators will be introduced according to the following organization:

* syntax
* list and type of interfaces: ? list of inputs, ! list of outputs
* informal description of the resulting effect
* constraints on the clocks of the input and output signals, written in terms of the clock algebra of the preceding paragraph.

3.2.1. The DELAY.

* x := y(n) $ m (†)
* ? y, of type T arbitrary; !x of type T^m (a vector of dimension n over T), where n and m are parameters of integer type
* writing y as follows

$$y_{-m-n+1},\ldots,y_0\ (\textit{initial conditions}),\ y_1,\ldots,y_t,\ldots(\textit{signal})$$

the signal x is defined by

$$x_t = (y_{t-m},\ldots,y_{t-m-n+1})$$

* !x ≡ ?y

COMMENT: the DELAY do not refer to any universal clock, but is nothing but a shift register with a specified length and memory, associated to a given signal. This differs from the usual real time languages, and also from ESTEREL (Berry & Cosserat (1984)), where the primitive instruction *wait* refers to a specified time unit.

(†) time is money, hence the notation.

3.2.2. The FILTER generator.

- x := y ↓ a
- ? y and a, of arbitrary types; !x, of same type as y.
- when both y and a are available, then y is delivered at the output x.
- ?y ~ ?a, !x ≡ ?y ∩ ?a.

COMMENT: this generator will be a component of the IF statement we shall provide to the programmer.

3.2.3. The MERGE generator.

- x := a ↑ b
- ? a and b, of the same type; !x, of the same type as a
- the signals a and b are merged, with a priority to a (i.e. b is lost when a and b are available simultaneously), and delivered at the output x
- ?a ~ ?b , !x ≡ ?a ∪ ?b

COMMENT: the FILTER and MERGE generators are sufficient for realizing the operations in the lattice of the signals dominated by a given one.

3.2.4. The SELECT generator.

- h := tt (c)
- ?c of boolean type; !h pure clock
- the occurrences "true" of c are delivered, whereas the occurrences "false" are lost.
- !h ≡ tt(?c), which implies h<c unless c is the constant "true" (††)

(††) this defines informally the map "tt" which associates to every boolean signal the clock of its occurrences "true"

COMMENT: together with the FILTER generator, this generator will be used to define the IF statement provided to the programmer.

3.2.5. The FUNCTIONS.

- *f* (generic notation)
- ?f, !f generic notation of the input and output signals, respectively.
- not specified
- !f = ?f for every input and output.

COMMENT: a typical example is the adder a := b+c. Note that, in this case, the constraint introduced on the clocks of the interfaces of any function *results in a strong synchronization of the input signals* (here a and b). This remark will be of great importance for the sequel.

Among the set of functions, we shall distinguish the *constant functions*, denoted by

const(x)

where the signal x is the input, and the output value has the constant value "const". Note that const(c) differs from tt(c) for a boolean signal c.

We have presented all the primitive temporal generators.

3.2.6. THE CLOCK CALCULUS OF A PROCESS.

To every process, we shall assign a set of equations of the clock algebra of its signals, and we shall refer to this set of equations as *the clock calculus* of the process. The purpose of this clock calculus is to verify the correctness of the synchronization mechanisms specified by the programmer, and to prepare the compilation. In order to illustrate this point, let us begin with the investigation of the following simple errored example.

An errored example

Consider the following program

P {{? **real** x,y ! **real** z }

boolean c, **pure clock** h}

P = (c := x<y ; h := **tt** (c) ; u := x $\downarrow$ h ; z := u+y) : z

end P

An informal description of this program is

"keep x only when x<y, and add the result to y"

The timing of this program is clearely errored for the following reason: the statement c:=x<y requires that x and y be simultaneous; then u is undersampled with respect to x; but z:=u+y requires again that u and y be simultaneous, which is clearely not satisfied for arbitrary values of the input signals x and y ! Obviously, the problem arises from the active role in the synchronization mechanisms played by the "functions". On the other hand, this active role is clearely necessary (what could be the meaning of z:=u+y if u only is available?).

This simple example justifies the necessity of a static verification of the timing of a SIGNAL program. Moreover, as we shall see later, this static verification will improve the efficiency of the execution of this program. The clock calculus will be the basic tool of this static verification.

A process is specified as follows:

P {{? list of inputs ! list of outputs }

synchro %set of clock equations%, other specifications }

P = %body%

end P

The body of P provides the compiler with

- a set of primitive generators resulting in the transformation of clocks, according to the rules described before
- primitive interconnection operators resulting in clock transfers.

Together with the set of equations given in the chapter "synchro", the whole results in a set of equations we refer to as *the clock calculus* which has to be solved by the compiler as we shall indicate later.

We shall now recall the clock equations caused by primitive generators, and indicate the clock transfers caused by the primitive interconnection operators.

3.2.7. Clock equations caused by generators.

We summarize them in the following table

CLOCK EQUATIONS CAUSED BY GENERATORS	
generator G	set of equations EQ(G)
x := y[n]$m	!x ≡ ?y
x := y ↓ a	?y ~ ?a , !x ≡ ?y ∩ ?a
x := y ↑ a	?y ~ ?a , !x ≡ ?y ∪ ?a
h := tt(c)	h ≡ tt(c)
function f	?(f) ≡ !(f)

The case of the constants: as a convention, constants have an unspecified

clock, but can be synchronized by functions as illustrated by the following examples:

x := y+z+1

stands for

x := y+z+1(y)

(recall that 1(y) is the signal which takes the constant value 1, and whose clock is the same as y), and is correct provided that y and z do not have different clocks. On the other hand

y := 3+7 ; z := x+y

gives z≡x≡y, since the assignment y:=3+7 do not specify any clock for y, the clock of y being (possibly) specified by the second generator. Unspecified clocks will be properly handled by the clock calculus.

3.2.8. Clock transfers caused by interconnection operators.

They are summarized in the following table; the corresponding justification will result from the formal semantics of SIGNAL we shall present in a forthcoming paper. In this table,

EQ(P)

denotes the clock calculus of the process P, i.e. the set of its clock equations, whereas

EQ(P){a/b}

denotes the clock calculus obtained by substituting b to a in the clock calculus of P.

CLOCK TRANSFERS
EQ(P ?a:x) = EQ(P){?a/?x}
EQ(P !b:y) = EQ(P){!b/!y}
EQ(P @x) = EQ(P){?x/!x}
EQ(P&Q) = EQ(P) ∪ EQ(Q)

COMMENTS: the rule of P@x reflects the fact that every output named x is connected to every input with the same name. The rule of P&Q expresses that the & results in the union of the corresponding clock calculi.

3.2.9. Some examples.

We shall not present here any formal result about the clock calculus (this will be the subject of a forthcoming paper), but we shall rather illustrate on simple examples how this calculus works.

Back to the errored example.

Recall the errored program we have introduced before.

P {{? real x,y ! real z } boolean c, pure clock h }

P = (c := x<y ; h := **tt** (c) ; u := x $\downarrow$ h ; z := u+y) :z

end P

First, the body of P is rewritten in terms of the primitive interconnection operators:

P = (c := x<y & h := tt(c) & u := x ↓ h & z := u+y) @ c,h,u : z

A bottom-top processing of the syntactic tree produces the following intermediates stages of the clock calculus:

EVOLUTION OF THE CLOCK CALCULUS	
current instruction	current state of the clock calculus
c := x<y	?x ≡ ?y ≡ !c
&	take union with the next equation
h := tt(c)	?x ≡ ?y ≡ !c, !h ≡ tt(?c)
&	take union with the next equation
u := x↓h	?x≡?y≡!c, !h≡tt(?c), ?x~?h, !u≡?x∩?h
&	take union with the next equation
z := u+y	?x≡?y≡!c, !h≡tt(?c), ?x~?h, !u≡?x∩?h, ?u≡?y≡!z
@c,h,u	?x≡?y≡!c, !h≡tt(!c), ?x~!h, !u≡?x∩!h !u≡?y≡!z

At this step, an error is detected, since the last set of equations imply

?x ≡ ?x ∩ tt(!c), !c ≡ ?x

which turns out to imply

?x < ?x

a contradiction.

Specification of a simple IF statement.

An informal description of this process is as follows: given the signals x, y, and a condition c (x, y, c simultaneous), select x or y according to c is true or false, and deliver the result at the ouput named z. This program is written as follows.

IF {{ ? x,y, bool c ! z } **synchro** ?x = ?y = ?c }

IF = (th:= **tt** (c) & fh:= **ff** (c) & u:=x ↓ th & v:=y ↓ fh & z:=u ↑ v)

@ th, fh, u, v : z

end IF

Here, **ff** (c) stands for short instead of **tt** (not c). The corresponding final clock calculus is the following

?x=?y=?c, !th=tt(?c), !fh=ff(?c), !u=?x∩!th, !v=?y∩!fh, !z=!u∪!v

which is a correct clock calculus. Using the rules

b≡h if (b≡x∩h and h<x)

tt(c)∪ff(c)≡c

we get the following final form with the minimal set of different clocks:

H=?x=?y=?c=!z , H'=!th=!u , H''=!fh=!v,

where

H'=tt(?c) , H''=ff(?c).

This is the typical desired form for the clock calculus of a process, namely

- *a list of clocks nested through conditions* (here H,H',H'' related through the condition c, with H as *master clock* , i.e. as the only free clock in the corresponding system of equations)
- *for every clock, the list of the signals belonging to it.*

A systematic study of such "normal forms" will be provided in a forthcoming paper.

Specification of a counter.

We shall present the specification of the macro "n counts x" we shall introduce in the next paragraph. The aim of this process is to count the past occurrences of a signal; the usefulness of such an instruction comes from the fact that the time is implicit in SIGNAL, so that it has to be explicitely generated when it is really needed. A possible program is as follows.

```
COUNT {{ ? x ! n } synchro !n=?x }
COUNT = ( n := zn+1 | zn := n $ 1 ) : n
end COUNT
```

This example illustrates the following fundamental remarks. First the process

```
n := zn+1 | zn := n $ 1
```

has only outputs as interfaces; as a consequence, *no clock can be assigned to the outputs of this process.* This is consistent with the fact that, once the initial condition is known, a counter can be fired infinitely many times. Hence, strictly speaking, the corresponding output data are all available at the initial instant "zero". Another consequence is that the body of the

process COUNT *does not involve the input signal x* ! The link between ?x and !n is concentrated in the specification of the synchronization of these signals, namely in the statement "synchro n=x", which specifies that both signals are simultaneous.

CONCLUSION: These examples illustrate the usefulness and the power of the clock calculus. Static verifications can be done on the timing of a SIGNAL program; at the same time, the number of different clocks can be reduced as much as possible, which will result in a dramatic reduction of the number of states and possible transitions when a finite state automaton is chosen as a target for the compiler (as for ESTEREL, Berry & Cosserat (1984), for example).

3.3. THE TEMPORAL GENERATORS PROVIDED TO THE PROGRAMMER, AND THE CORRESPONDING INSTRUCTION SET OF THE LANGUAGE SIGNAL.

Rather than to provide the primitive temporal generators to the programmer, we preferred to provide to him higher level instructions, derived as macros from the primitive ones. These instructions will be organized in 4 chapters:

* the delay
* the purely synchronizing instructions
* the IF statement
* the MUX statement.

They will be presented according to the following scheme

- syntax and types of labelled ports
- informal description
- formal definition as a macro.

3.3.1. The delay.

- syntax

x is y(n) $m

?y of arbitrary type *T* !x vector of dimension n over *T*;

(n) is optional: "x is y $m" stands for "x is y(1) $m".

- informal description

x is the m-step delayed sliding window of dimension n over y.

- formal description

The delay is primitive, see the corresponding paragraph, where it was denoted by "x := y(n) $m".

COMMENT: The reason for introducing the new notation "x is..." instead of the operator-like notation "x :=..." is precisely that we want to avoid the delay to be used as an operator inside an expression. Operator-like instructions (such as +, *, and, or,...) will be allowed for describing only *functions*, i.e. memoryless transforms of signals. On the other hand, temporal instructions will generally be constrained to be used as *processes*, which is the case for the present expressions of the delay.

3.3.2. Purely synchronizing instructions.

These instructions are useful for the following reasons. First, in SIGNAL, values never are persistent; this is a drawback when a past value has to be

kept available for some unspecified time (like variables). Second, the time is implicit, which can also be a drawback, if a modification of some task at a specified instant is searched for. Hence the following instruction set.

EXTRACTS

* syntax

y extracts x at b

?x,b compatible (?x~?b), of arbitrary types; !y of same tpe as ?x

* informal description

y delivers the current value of x when both x and b are available (hence the values of the signal b are irrelevant for this instruction).

* formal description.

This instruction is nothing but the primitive generator FILTER of the preceding paragraph.

EXTENDS

* syntax

y extends x at b

?x,b compatible, of arbitrary types !y of the same type as x.

* informal description

The last received value of x is delivered at the output y when x or b is available; this is the basic instruction for constructing variable-like signals.

* formal definition as a macro

y extends x at b {{?x,b !y} synchro ?b~?x, !y=?b∪?x }

y extends x at b = (y:=x↑zy | zy is y $1):y

end

COMMENT: again, let us emphazise on the key role played by the synchro specification, since the body of the process does not involve the input b, and does not specify completely the clock of the output y (the only consequence of this body is the inequality !y≤?x). This instruction allows the programmer to build variable-like signals.

The COUNTERS.

There are three counters.

* syntax

(a) n counts y

(b) n counts y from a

(c) n counts y after a

?y,a of arbitrary types !n of integer type

* informal description

(a): n counts the past occurrences of the signal y

(b): n counts the occurrences of y which are not anterior to the last occurrence of a

(c): n counts the occurrences of y which are posterior to the last occurrence of a.

* formal definition as macros.

n counts y {{?y ! integer n} synchro n≡y}

n counts y = (n := zn+1 | zn is n $1):n

end

n counts y after a {{?y,a ! integer n} synchro y~a, n = y∪a}

n counts y after a = (n:=zn+1 | znn is n $1 | zn:= -1(a)↑znn):n

end

Recall that -1(a) denotes the signal whose clock is the clock of a, and whose value is the constant value -1.

n counts y from a {{?y,a ! integer n} synchro y~a, n=y∪a}

n counts y from a = (n:=zn+1 | znn is n $1 |

((zzn := -1(a)↑znn & x := y↓a); u := 1(x)↑0(a); zn:=zzn+u)):n

end

SELECT

* syntax

h selects %boolean expression%

? signals involved in the boolean expression (a and b if bool exp = a<b, for example), !h pure clock signal.

* informal description

the pure clock signal h selects the instants where the boolean expression can be evaluated, and is true (the signals involved in the boolean expression do not need to be simultaneous, but only compatible).

* formal description as a macro

Denote by boolexp(a,b) a boolean expression involving the signals a and b.

h selects boolexp(a,b) {{?a,b ! pure clock h} synchro a~b}

h selects boolexp(a,b) = ((aa extracts a at b & bb extracts b at a);

c := boolexp(aa,bb) ; h:=tt(c)):h

end

This macro definition can be obviously extended to boolean expressions involving an arbitrary number of signals.

3.3.3. The IF statement.

As we shall see, the IF statement do not act on signals but on processes.

* syntax

if c then P else Q fi

c is a boolean signal or a condition, P and Q are processes

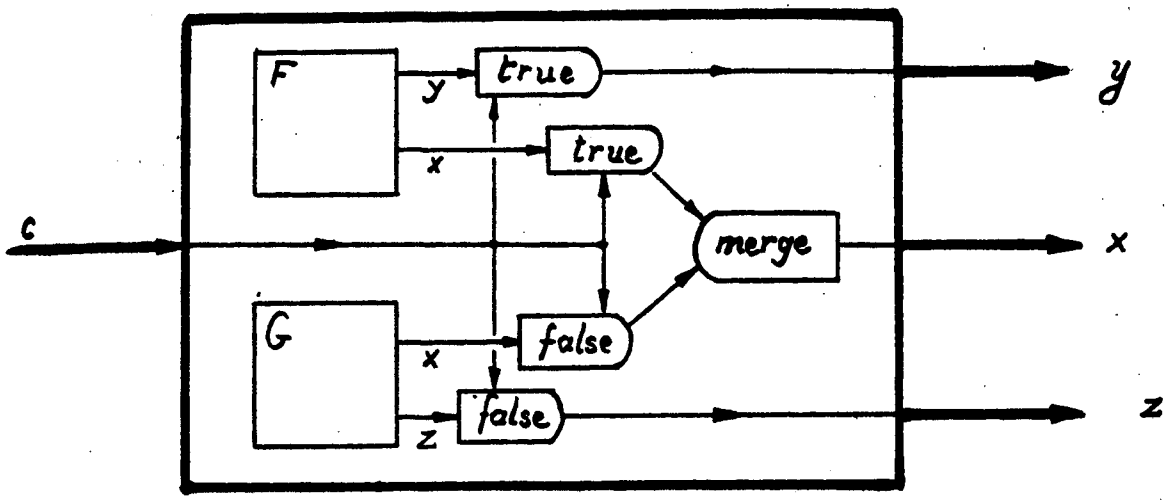
? = c∪?P∪?Q ,! = !P∪!Q where, as before, ?P denotes the set of the outputs of the process P.

"then P" or "else Q" are optional.

* informal description

The processes P and Q are always active, independently of the condition c; the IF statement acts on the outputs of the processes P and Q in the following way. Given an output port of P (resp.Q), the value carried by this port at a given instant is delivered at the output of the resulting process if and only if c is available and true (resp. false); if two output ports of P and Q have the same name, their filtered signals are merged to give a single port of the resulting process. This is summarized in the diagram below.

fig.(3.1): IF statement, graphical example



- formal definition as a macro (corresponding to the diagram above)

if c then P else Q fi {{? bool c, ?P,?Q !x,y,z} synchro c ~ !(P) ~ !(Q)}

if c then P else Q fi =

(((P;(y extracts y at select c & x1 extracts x at select c))&

(Q;(z extracts z at select not c & x2 extracts x at select not c)));

x := x1 ↑ x2):x,y,z

end

two typical examples are:

if x>0 then y:=x else y:=-x fi

if x>0 then y:=x fi

In the former example, the result is simply y:=|x| with the same clock as x, whereas in the second example, the process delivers x only when x is positive, and nothing in the other case, so that y is undersampled with respect

to x.

3.3.4. Synchro statements.

These statements are introduced in the specifications of the process, as we have seen many times in the definitions of the above macros. These synchro specifications are expressed in terms of the clock algebra we have introduced; in order to cope with the problem of the visibility of the names of the signals internal to a given process (a single name can represent different signals at different locations in the body of the process), *such "synchro" statements can involve only interfaces of the process.* Although we used many times such synchro statements in the definition of the macros, their use is expected to be very seldom, thanks to the instruction set we have provided to the programmer.

3.3.5. The Multiplex.

We shall not give any instruction for the multiplex in this report; further studies are required to define properly what is the clock of a multiplexed signal; this will done in a forhtcoming paper.

We shall now give a summary of the temporal instruction set of SIGNAL.

TEMPORAL INSTRUCTIONS		
name	syntax	effect
DELAY	y is x(n) $m	see 3.3.1
IF	if c then P else Q fi	see 3.3.3
EXTRACTS	y extracts x at a	see 3.3.2
EXTENDS	y extends x at a	see 3.3.2
SELECTS	h selects "boolexp"	see 3.3.2
COUNTERS	n counts x (from/after) a	see 3.3.2

4. THE DEPENDENCE GRAPH OF A SIGNAL PROGRAM.

The aim of this chapter is to introduce the reader to some subtleties related to this dependence graph. Some of the problems we shall encounter have been investigated in a formal manner in Berry & al. (1984) for the language ESTEREL; the corresponding formal study for SIGNAL will be the subject of a forthcoming paper. Our current purpose is only to illustrate these difficulties.

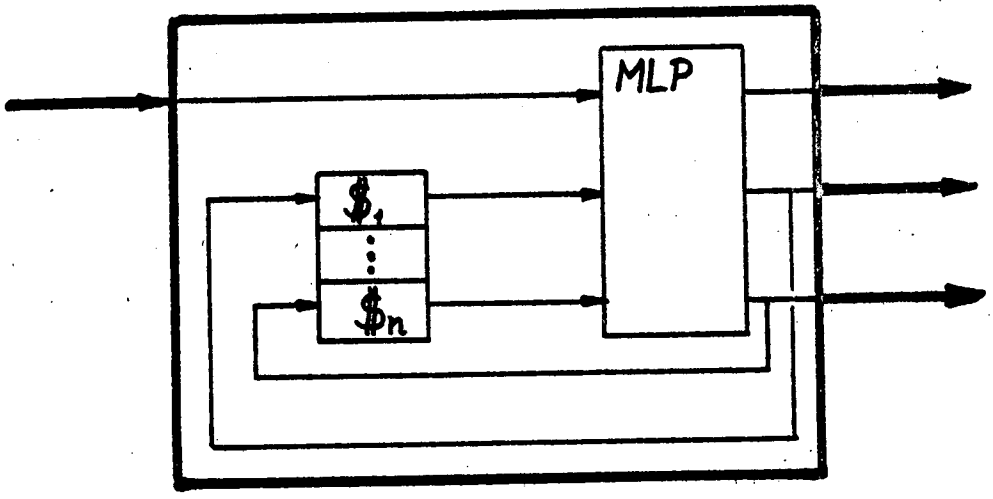
4.1. The case of a SIGNAL process with a single clock.

This corresponds in fact to a wellknown situation where the only temporal generators are delays. The derivation of the associated dependence graph is straightforward and wellknown in both areas of computer science and network theory; we refer the reader to the classical book of Oppenheim

& Schafer (1976) for a classical treatment in the framework of the network theory. Roughly speaking, the derivation of this dependence graph is as follows.

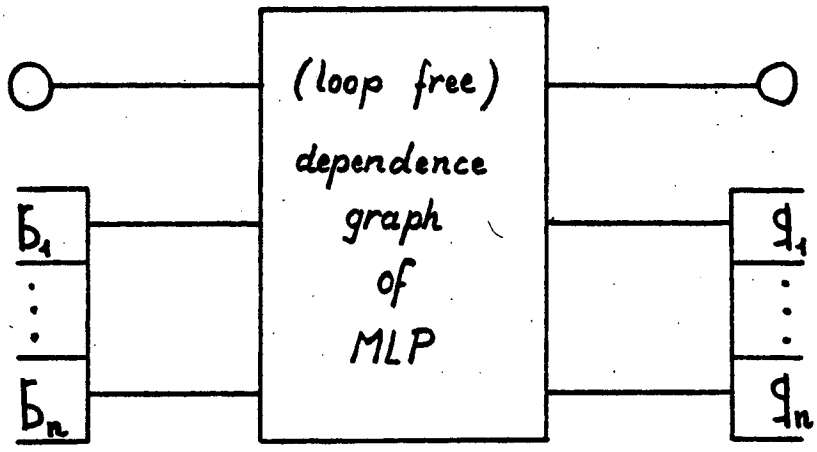
First, reorganize the static network described in the SIGNAL program as depicted in the following figure.

fig.(4.1): a process with a single clock



In this figure, the $'s represent a set of DELAY generators in parallel, i.e. of fifo registers; the main block named MLP (for MemoryLess Process) involves only *functions* , i.e. instantaneous transformations of the signals. Then the dependence graph is simply obtained by replacing every $ by two nodes: the first one delivers the past value of the corresponding signal (and thus has the level *zero* in the precedence graph, exactly as the inputs of the process), whereas the second one delivers the current value of the same signal and has the level in the precedence graph corresponding to the production of this value. The result is shown in the following figure.

fig.(4.2): the corresponding dependence graph.



The automatic production of the dependence graph from the SIGNAL program is straightforward, and is not described here.

Unfortunately, the derivation of the dependence graph is much more involved when the SIGNAL program involves several clocks. A complete study of this problem will be presented in a subsequent paper; we shall only illustate the difficulties in a simple example.

4.2. Derivation of the dependence graph when several clocks occur in the SIGNAL process: an example.

Consider the following SIGNAL program:

P {{**? real** y **!real** x } **par real mu**}

P = (**if** y>mu **then** y':=y **fi**; (x:=.5*zx+y' | zx **is** x **$1**)): x

where

real zx,y'

end P

This program selects the instants where y>mu, and generates at the corresponding clock the recurrent equation x := .5*x + y. It is apparent on this example that it is not possible to assign to the signal zx the level zero in the corresponding dependence graph, since the instants at which this signal has to be delivered depends on the values of the input signal y. To present a convenient solution, we shall detail how the clock calculus and the dependence graph are simultaneously obtained.

First, the body of the program is rewritten in terms of the primitive operators.

(if y>mu **then** y':=y **fi** ; (x:=.5*zx+y' | zx **is** x .B $1)): x

=

((h:= tt (y>mu) & y':=y $\downarrow$ h & x:=.5*zx+y' & zx **is** x $1) @ h,x,y',zx): x

To derive the dependence graph, the following rules are applied.

- for every generator, except the delays, a directed branch is assigned to every input output pair, which originates at the input node and terminates at the output node.
- the effect of the connection operators are the same as for the clock calculus (hence the simultaneous derivation of the dependence graph and the clock calculus.)
- the input of a DELAY generator *is not a source node* of the dependence graph, but a directed branch is created which terminates at this node, and originates at the node of the clock of the interfaces of this DELAY; to this branch is assigned the label " $ ".

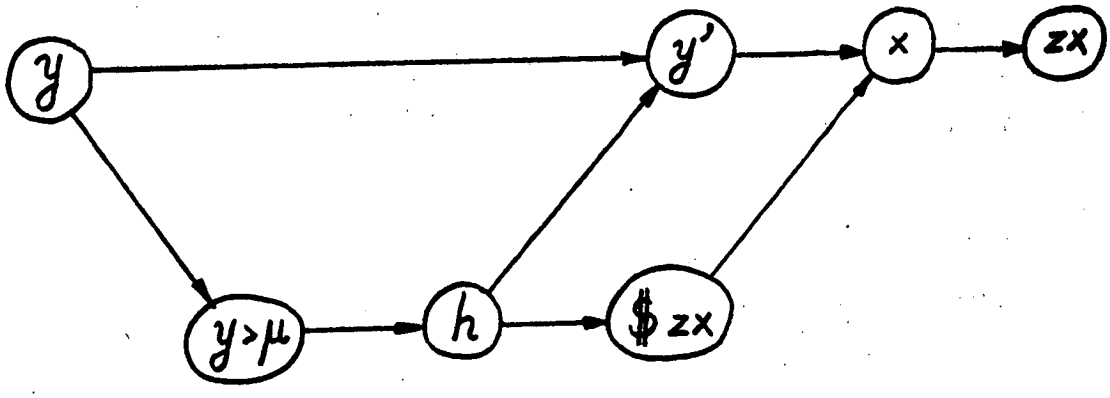
- to every clock is assigned a node in the graph. According to the previous analysis, a clock is either the clock of an input signal, or created by a condition. In the former case, a directed branch originates at the input signal and terminates at the corresponding clock. In the later case, the dependence is created by applying the first rule to the generator "tt()", and to the corresponding branch is assigned the label " tt ".

The following table describes the simultaneous derivation of the dependence graph and the clock calculus in the former example. In this table, a directed branch of the dependence graph is denoted by {originating node, terminating node}, whereas a marked branch is denoted by { , ;"$" or "tt"}.

DEPENDENCE GRAPH AND CLOCK CALCULUS		
current instruction	current graph	current clock calculus
h:=tt(y>mu)	{(?y>mu),!h;tt}	!h=tt(?y>mu)
y':=y-h	{?h,!y'} {?y,!y'}	!y'=?h ?y~?h
x:=.5*zx+y'	{?y',!x} {?zx,!x}	?y'=?zx=!x
zx is x $1	{?x,!zx;$}	!zx=?x
(& & &)	take the union of the preceding subgraphs	take the union of the preceding subcalculi
@h,y',x,zx	{(?y>mu),!h;tt} {!h,!y'} {?y,!y'} {!y',!x} {!zx,!x} {!x,!zx;$}	!h=tt(?y>mu) !y'=!h !y'=!zx=!x

To obtain the final graph, the branch labelled "$" has to be modified as follows: its terminating node remains unchanged, but its origin is replaced by the node of *the clock of the corresponding signal* , i.e. the clock of !x, namely !h. The figure (4.3) shows the corresponding dependence graph.

fig.(4.3): an example of dependence graph for a multiple clock process.



The complete method will be presented in a forthcoming paper, where the formal semantics of SIGNAL will be detailed.

5. EXAMPLES.

In this chapter, we shall try to illustrate both aspects of the language SIGNAL, namely the construction of the static networks, and the synchronization mechanisms. Moreover, this will be the opportunity to introduce the reader to the structure of SIGNAL programs.

5.1. Concurrent matrix multiplication.

This is a wellknown classroom example, which was used by numerous authors, see H.T.Kung (1982), the WAP of S.Y.Kung & al. (1982); this presentation follows P.Quinton (1984).

Consider two n×n matrices A and B, and let C = AB. Then the matrix product

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}, 1 \le i,j \le n \qquad (5.1)$$

can be rewritten as follows

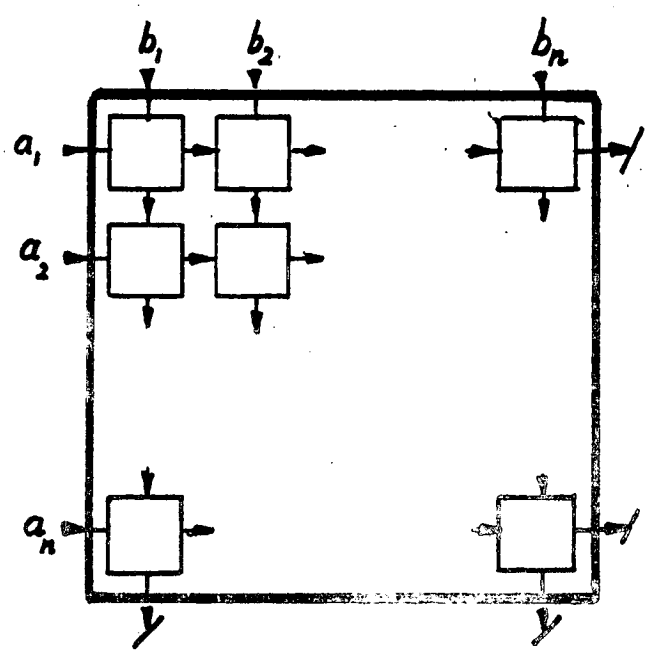
$$c_{ij}(k) = c_{ij}(k-1) + a_i(k)\, b_j(k) \qquad (5.2)$$

where a_i and b_j are respectively the i-th row of A and the j-th column of B. This form suggests

- to consider the index k as a time index
- to assign a single process to the computation of every c_{ij} coefficient.

The result corresponds to the so-called *Uniform Recurrent Equations*. (Quinton (1984)), which are depicted in the following diagram.

fig. (5.1): matrix multiplication.



We shall now describe the corresponding SIGNAL program. According to the equation (5.2), we shall consider the i-th row a_i of the matrix A as a *scalar signal* whose k-th occurrence is $a_i(k) = a_{ik}$. The value of the c_{ij} coefficient of the matrix C will be the n-th occurrence of the signal c_{ij}, as computed by the equation (5.2).

```
MULT {{ ? [n] real A,B ! [n,n] real C } par int n }
=
( doseq i until n of
    ( doseq j until n of NODE ? b:B[j] ! b:B[j],c:C[i,j] od)
    ? a:A[i] ! a:A[i]
od)
:C

where

process
    NODE {{? real a,b ! real a,b,c}}
    NODE =
    (k counts a & (c := zc + a*b | zc is c $1)/zc);
    c extracts c at k=n
    where
        real zc, int k
    end NODE

end MULT
```

Let us comment this program (%...% denotes comments). Its general structure is as follows.

```
PROCESS-NAME
    {
        {list of interfaces}
        type of interfaces, and related parameters
        synchro specifications on the interfaces
        }
    =
    % body of the process %
    where %the ordering of the forthcoming paragraphs is arbitrary%
        % types of internal signals %
        % parameters, such as initial conditions to start up delays,... %
        process
            SUBPROCESS-NAME {{ interfaces } local specif. related to
            interfaces }
            SUBPROCESS1 = % body %
            SUBPROCESS2 = % body %
            % SUBPROCESSi denotes different processes with the same
            interfaces specified in SUBPROCESS-NAME {{}} %
    end
```

In the example above, the body of the main program MULT.

together with the knowledge of the interfaces of the black box NODE, entirely specifies the rectangular array depicted in the figure (5.1). On the other hand, the subchapter "process" in the chapter "where" specifies the inside of the black box NODE. This structure is suitable for a hierarchical description of signal processing algorithms.

The program we have presented specifies the matrix multiplication in terms of the Uniformly Recurrent Equations, according to the framework of Quinton (1984). To get an implementation of this algorithm on the Wavefront Array Processor (S.Y.Kung & al. (1984)), considered as a virtual machine, this program is modified as follows.

```
MULTWAP {{?(n) real A,B !(n,n) real C} int par n}
=
((dopar j to n of (B[j] is B[j] $j-1));
(doseq i to n of
    (A[i] is A[i] $i-1 ;
    doseq j to n of
        (NODE; (a is a $1 & b is b $1)) ?b:B[j] !b:B[j],c:C[i,j]
    od ?a:A[i] !a:A[i])
od))
:C
where % same as before %
```

The delays appearing in (NODE;...) describe the pipelining, whereas the other delays appearing in the cascades "dopar" and "doseq" synchronize the inputs at the boundaries of the array. For the *initializa-*

tion of the delays, we applied the following implicit rule:

- a delayed signal can be initialized in the "where" section of the program;
- if no initialization is given, an implicit initialization to the value "zero" is assigned for a real (or integer, or complex,...) signal.

A major difference with the pure data flow approach (as used in the WAP by Kung & al (1982)) is that additional delays are required to explicitely synchronize the *phases* of the clocks of the input signals; such a synchronization is automatically realized in the pure data flow approach (a counterpart is the lack of synchronism in pure data flow approaches).

5.2. On-line detection of jumps in the mean of a signal, using Page's stopping rule

See Basseville & Deshayes (1984) for the theory behind such algorithmes. This algorithm is a prototype of processing required for the monitoring of signals or systems for the purpose of fault detection or pattern recognition. This example has been chosen for to illustrate the synchronization mechanisms.

An informal description of this algorithm is as follows. Denote by (y_t) the signal to be monitored.

Step 1: for t>0 do

$$m_t = m_{t-1} + g*(y_t - m_{t-1})$$

Step 2: for t≥tmax do in parallel

$$S_t^+ = S_{t-1}^+ + (y_t - m_t - jmin)$$

$M_t^+ = max(S_s^+ : s \leq t)$

$T_t^+ = arg\ \max(S_s^+ : s \leq t)$

test: $M_t^+ - S_t^+ > \mu$?

and

$S_t^- = S_{t-1}^- + (y_t - m_t - jmin)$

$M_t^- = min(S_s^- : s \leq t)$

$T_t^- = arg\ \min(S_s^- : s \leq t)$

test: $S_t^- - M_t^- > \mu$?

Step 3: when test=true deliver T^+ or T^- according to which test was positive, and go back to step 1.

The two thresholds are *jmin* (minimum size of the jump to be detected) and *mu* (threshold of the test); m_t is the current estimate of the mean; $M^+ - S^+$ (and the other with "-") are the loglikelihood ratios of the alternative hypothesis *a negative jump occured (resp. positive)* against the null hypothesis *no jump occured.*

This example exhibits few parallelism, so that it is easely described using a conventional sequential language like PASCAL. Nevertheless, our purpose is here to illustrate how the timing mechanisms describe the different clocks running the signals. The SIGNAL program is as follows:

```
PAGE'S TEST ( {? real y ! real m; int T} par real jmin,mu,g ; int tmax )
=
(n counts y;
```

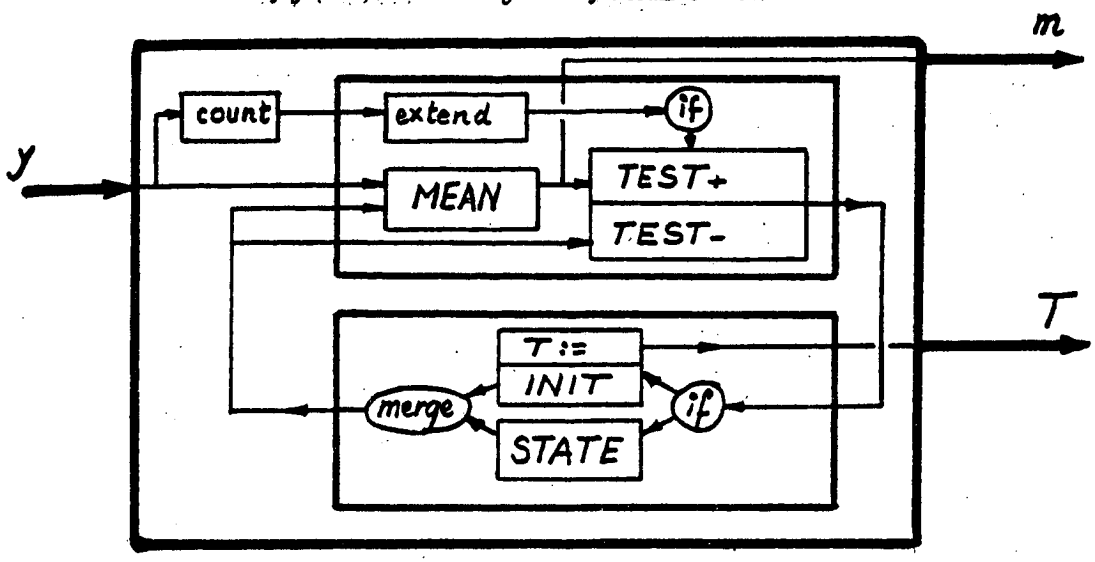
```
(((MEAN & TT extends T at y); if n-TT>tmax then TEST+ & TEST- fi)|
if (M1 - S1 > mu) or (S2 - M2 > mu)
    then INIT & if M1 - S1 ≥ mu then T:=T1 else T:=T2 fi
    else STATE
fi)
):m,T

where

int n,TT,T1,T2

process

    MEAN {|? real y,zm ! real m|}
    MEAN =
    m := zm + g*(y-zm)
    end MEAN

    TEST+ {|? real m,y,zS1,zM1; int n,zT1 ! real S1,M1; int T1 |}
    TEST+ =
    S1 := zS1 + (y-m+jmin);
    if S1 ≥ zM1
        then M1 := S1 & T1 := n
        else M1 := zM1 & T1 := zT1
    fi
    end TEST+
```

```
TEST- {{? real m,y,zS2,zM2 ; int n,zT2 ! real S2,M2 ; int T2 }}
TEST- =
S2 := zS2 + (y-m-jmin);
if S2 ≤ zM2
    then M2:=S2 & T2:=n
    else M2:=zM2 & T2:=zT2
fi
end TEST-

INIT {{? int n ! real zm,zS1,zS2,zM1,zM2; int zT1,zT2 }}
INIT =
zm:=0 & zS1:=0 & zS2:=0 & zM1:=0 & zM2:=0 & zT1:=n & zT2:=n
end INIT

STATE {{? real m,S1,S2,M1,M2 ; int T1,T2
    ! real zm,zS1,zS2,zM1,zM2 ; int zT1,zT2 }}
STATE =
zm is m $1 &
zS1 is S1 $1 & zS2 is S2 $1 &
zM1 is M1 $1 & zM2 is M2 $1 & zT1 is T1 $1 & zT2 is T2 $1
end STATE

end PAGE'S TEST
```

As in the previous example, the statement "where" is used for a hierarchical description of the process. For example, the main pro-

gram describes the block diagram depicted in the figure (5.2).

fig.(5.2): block diagram of PAGE'S test.



This figure shows clearely the initialization procedure: the signal flows through INIT instead of through STATE. The pathes connecting the black boxes carry signals with different clocks: for example the output of the block TEST+ & TEST- is undersampled with respect to the global input y, whereas the global output T is delivered only when a change has been detected, which corresponds also to an undersampling. The clock calculus associated to such an example is rather complex. More complex algorithms, such as used in some TV coding schemes (see Richard & al (1984) for an example), can involve more than 100 different clocks, thus requiring a tremendeous effort to describe them in a conventional language like FORTRAN or PASCAL; in such cases, the synchronization mechanisms of SIGNAL reduce this effort in a dramatic way.

However, we should point out the following. If the programmer is interested in restarting the monitoring procedure at the estimated instant of change, rather than at the instant of detection, then the

corresponding algorithm *can no more be described* in SIGNAL. The reason is that, to implement this algorithm, a buffer of *unspecified* length is required, so that this algorithm is no more a real time algorithm as defined in the introduction, and hence, cannot be described in SIGNAL.This restriction is at the same time a drawback and an advantage, since a SIGNAL specification is a guaranty for an algorithm to be real time.

6. CONCLUSION

We have presented the language SIGNAL, which is a data flow oriented, real time, synchronous, side effect free language suited to the expression and recovery of the parallelism in signal or image processing real time algorithms. This language is intended to be an entry point of a CAD chain for the implementation of complex signal processing tasks on multiprocessor architectures.

The main contributions of this new language are the following:

- a static (i.e. before running the program) verification of the correctness of the timing is performed by the SIGNAL compiler.
- Starting from the SIGNAL program, the compiler is able to derive the corresponding dependence graph, which is of primary interest when the ultimate objective is the implementation on a multiprocessor architecture. Let us emphazise that the *clock calculus* we have introduced for the verification of the timing is in fact a part of this dependence graph.

An interesting result is that a correct SIGNAL program can be considered as a virtual *single token pass data flow machine*; here, "virtual" means that every action in this machine has a zero duration.

The version of the language SIGNAL we have presented here should be considered as a *v0* version, i.e. as a preliminary one. The first complete version will be the *v1* version, which will include the MUX (multiplexing) operator, thus allowing the oversampling of signals. The version *v1* will allow the programmer to specify (and execute) any real time task relevant to signal or image processing. This claim will be supported by the currently developped formal semantics of the language SIGNAL. The version *v1* of the language will be oriented to implementation. For example, we intend to provide to the programmer parametrized elementary functions to simulate rounding effects.

Because of the constraint we imposed on the algorithms to be described, namely to be real time, it will be desirable to extend the language to a non real time version *v2* oriented to pattern recognition. This will be the subject of a future research.

REFERENCES

Ackerman (1979) : W.B. Ackerman,"data flow languages", *Proc. AFIPS conf. (E.Mervin Ed.)*, New York.

Ashcroft (1977) E.A. Ashcroft,W.W. Wadge "Lucid, a Nonprocedural Language with Iteration", *Comm. of ACM 20 (7) , pp 519-526.*

Backus (1978) : J.Backus,"Can Programming Be Liberated from the Von Neumann Style ? A Functional Style and its Algebra of Programs", *Comm. of the ACM 21 (8)* August 1978, pp 613-641.

Basseville & Deshayes (1984) : M. Basseville, J. Deshayes,"Detection of abrupt changes in signals and systems", *CNRS conf. Paris* 21-22 March 1984.

Berry & al (1983) : G. Berry, S. Moisan, J.P. Rigault,"Towards a synchronous and semantically sound high level language for real time applications", *rep. centre de Math. Appl. Sophia Antipolis.*

Berry & Cosserat (1984) : G. Berry, L. Cosserat,"The ESTEREL Synchronous Programming Language and its mathematical Semantics", *Rapport de recherche No 327, INRIA*, Sophia-Antipolis, Septembre 1984.

Caspi & Halbwachs (1982) : P. Caspi, N. Halbwachs,"Algebra of events : A model for parallel and real time systems," *Proc. of the 1982 int. conf. on parallel processing*, 1982, pp 150-159

Cremers & Hibbard (1978) : A.B. Cremers, T. Hibbard,"Formal Model of virtual machines", *IEEE Soft. Eng. 4, pp 426-436*

Dennis (1974) : J.B. Dennis, J.B. Fosseen, J.P. Linderman, "Data Flow Schemas", *Int. Symp. on Theory Programmation*, LNCS 5, Springer-Verlag, 1974 pp 187-216

Elmqvist (1978) : H. Elmqvist,"Dymola - A Structured Model Language for Large Continuous
Systems," *Summer Computer Simulation Conf.*, Toronto,Canada,July 1979.

Gautier (1984) : T. Gautier,"SIGNAL,a a data flow oriented language for signal processing," *IRISA Tech. Report.*

Hankin (1981) : C.L. Hankin, H.W. Glaser,"The Data Flow Programming Language CAJOLE -An Informal Introduction", *SIGPLAN Notices 16 (7)*, 1979ly 1981, pp 35-44.

Kahn (1974) : G. Kahn,"The semantics of a simple language for parallel programming", *Information Processing 74*, North-Holland Publ. Comp. ,1974, pp 471,475.

H.T. Kung (1982) : H.T. Kung, "Why Systolic Architectures ?", *Computer*, Vol. 15, No 1, Jan 1982, pp. 37-46.

S.Y. Kung & al (1982) : S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, D.V. Bhaskar Rao,"Wavefront Array Pro cessor : Language, Architecture, and Applications", *IEEE Transactions on Computers*, Vol C-31, No 11, Nov 1982, pp 1054-1066.

Le-Guernic (1982) : P. Le-Guernic, "SIGNAL : Description algebrique des flots de signaux", *Architecture des machines et systemes informatiques.Afcet*, 17-19 nov. 1982, Ed. Hommes et Techniques, pp243-252.

Le_Guernic (1883) : P. Le_Guernic, A. Benveniste, T. Gautier, "SIGNAL : Un langage pour le traitement du signal", *BIGRE 33*, Mars 1983, pp12-42.

Mc Carthy (1960) : J. Mc Carthy,"Recursive Functions of Symbolic Expressions and their Computation by Machine. Part 1", *Comm. of the ACM # (4)*, April 1960, pp 184-195.

Mc Graw (1982): J.R. Mc Graw,"The VAL Language : Description andd Analysyis", *ACM Trans. on Prog. Languages and Systems 4 (1)* Janvier 1982, pp 44-82.

Oppenheim & Schafer (1976) : A.V. Oppenheim, R. Schaffer,"Digital Signal Processing," *Prentice-Hall*, 1975.

Quinton (1984) : P. Quinton,"Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations", *Proc. 11'th Annual Symposium on Computer Architecture*, 1984, pp 108-214.

Richard & al (1984) : C. Richard, A. Benveniste, F. Kretz, "Recursive estimation of edges in TV pictures as applied to ADPCM coding", *IEEE Com. 32 No 6.*

Young (1982) : S.J. Young, "Real time languages: design and development", *Elis Horwood publishers*, 1982.

PI 234 **Architectures systoliques pour la reconnaissance de mots connectés (en anglais)**
François Charot, Patrice Frison, Patrice Quinton, 40 pages ; Août 1984.

PI 235 **A scheme of Token Tracker**
Zhao Jing Lu, 62 pages ; Septembre 1984.

PI 236 **Design of one-step and multistep adaptive algorithms for the tracking of time varying systems**
Albert Benveniste, 40 pages ; Septembre 1984.

PI 237 **The design and building of Enchère, a distributed electronic marketing system**
Jean-Pierre Banatre, Michel Banatre, Guy Lapalme, Florimond Ployette, 38 pages ; Septembre 1984.

PI 238 **Algorithme optimal de décision pour l'équivalence des grammaires simples**
Didier Caucal, 48 pages ; Septembre 1984.

PI 239 **Detection and diagnosis of abrupt changes in modal characteristics of nonstationary digital signals**
Michèle Basseville, Albert Benveniste, Georges Moustakides, 26 pages ; Octobre 1984.

PI 240 **Convergence optimale de l'algorithme de «réallocation-recentrage» dans le cas le plus pur**
Israël-César Lerman, 39 pages ; Octobre 1984.

PI 241 **Un algorithme d'exclusion mutuelle pour une structure logique en anneau**
Michel Raynal, 12 pages ; Novembre 1984.

PI 242 **Note sur l'interprétation de primitives d'action proximétriques en termes de liaisons cinématiques fictives**
Bernard Espiau, 20 pages ; Novembre 1984.

PI 243 **Robust detection of signals - A large deviations approach**
Georges V. Moustakides, 16 pages ; Novembre 1984.

PI 244 **Algorithmes de génération de caractères**
Gérard Hégron, 24 pages ; Novembre 1984.

PI 245 **Optimal stopping times for detecting changes in distributions**
Georges V. Moustakides, 10 pages ; Janvier 1985.

PI 246 **Signal : a data flow oriented language for signal processing**
Paul Le Guernic, Albert Benveniste, Patricia Bournai, Thierry Gautier, 62 pages ; Janvier 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique