Real-time,synchronous,data-flow programming: the language "SIGNAL" and its mathematical semantics

Paul Le Guernic, Albert Benveniste

UNITÉ DE RECHERCHE
INRIA-RENNES

Rapports de Recherche

N° 533

REAL-TIME, SYNCHRONOUS, DATA-FLOW PROGRAMMING: THE LANGUAGE "SIGNAL" AND ITS MATHEMATICAL SEMANTICS

Paul LE GUERNIC
Albert BENVENISTE

Juin 1986

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

REAL-TIME, SYNCHRONOUS, DATA-FLOW PROGRAMMING: THE LANGUAGE «SIGNAL» AND ITS MATHEMATICAL SEMANTICS.

Publication Interne n° 298
112 pages

June 1986

Paul Le Guernic, Albert Benveniste

IRISA/INRIA, Campus de Beaulieu
F35042 RENNES CEDEX, FRANCE

Abstract: *We present the kernel of the language SIGNAL. SIGNAL is a data-flow, real-time, synchronous language which was primarily designed for real-time control and signal processing task specification and implementation. Unlike classical data-flow or real-time languages, SIGNAL: is based on a synchronous notion of time. The semantics is of operational style, and allows to derive a complete static calculus of the timing of every SIGNAL process, called its clock calculus. Hence, the programming language SIGNAL is also a formal system to reason about timing and concurrency.*

Résumé: *Cet article présente le langage SIGNAL. SIGNAL est un langage de type flot de données, temps-réel, synchrone, dont le domaine d'application primitif est la spécification et la mise en oeuvre de tâches temps-réel dans les domaines du traitement du signal et de l'automatique. Contrairement aux langages classiques temps-réel ou flot de données, SIGNAL utilise un temps logique de type synchrone. La sémantique est de type opérationnel, et permet de fonder un calcul statique complet du comportement temporel d'un processus SIGNAL: son calcul d'horloge. Le langage de programmation SIGNAL est donc aussi un système formel permettant de manipuler simultanément le temps et le parallélisme.*

Paul LE GUERNIC

Albert BENVENISTE

REAL-TIME, SYNCHRONOUS, DATA-FLOW PROGRAMMING : THE LANGUAGE SIGNAL AND ITS MATHEMATICAL SEMANTICS

Publication interne
n° 298

Juin 1986

Table of Contents

Chapter One
INTRODUCTION.

The purpose of the language SIGNAL is to take place as an entry point in the two chains

{ high-level task specification } → { VLSI implementation }

{ high-level task specification } → { distributed system implementation }

in the areas of real-time signal or image processing, real-time control systems, and, more generally, C^3-type applications.

To achieve these goals, SIGNAL has been designed to be at the same time

- an executable language
- a formal system to reason about timing
- a formal system to reason about parallelism and concurrency.

Hence, the core of the language SIGNAL is based on a formal model; the purpose of this report is to present simultaneously the kernel of SIGNAL, the mathematical model it is based upon, and the formal proof system it provides on the above mentioned aspects. The detailed discussion about the characteristics of the selected fields of application, together with the programming style of the language, can be found in [Le Guernic & al. 1986], [Le Guernic & al. 1985].

SIGNAL is related to two classes of languages: the real-time languages, and the data-flow languages. Classical real-time languages, such as ADA [ADA 1980], LTR [LTR 1978] are basically asynchronous even if they provide explicit synchronisation mechanisms, and consider only one notion of absolute time reference (the "physical" time), hence their non deterministic character is impossible to control in a formal way. Again is the language OCCAM [OCCAM 1983] fully asynchronous in nature, but its rendez-vous mechanism is directly derived from the mathematical model CSP [Hoare 1978, Brookes & al. 1984], so that the guaranteed properties of CSP are still found in OCCAM whatever its (correct!) machine implementation is.

Data-flow programming is now a recognized way to ensure *functional* execution of a program on a machine with distributed memory and control [Ackermann & Dennis 1979, Dennis 1974]. Again is the execution fully asynchronous in nature, but the functional behaviour is guaranteed thanks to the following rule of executability of an operation: *a data-flow actor can fire when it has data tokens on all its input arcs, it then produces result token(s) on its own output arcs.* Unfortunately, the data-flow control machanism creates an overhead when it has to be implemented on a

classical Von Neumann architecture; and this can be the case when micro-actors of macro data-flow architectures are Von Neumann oriented; see [Chase 1984, Gaudiot 1985] for fully data-flow oriented architectures. Models of data-flow computing have been studied by Kahn [Kahn 1974, Kahn & Mc Queen 1977] using the notion of Dynamic Network Processes, see also [De Bruin & Boehm 1985].

SIGNAL is a *synchronous data-flow* language, a notion which will become clear throughout this article. A consequence is that *SIGNAL programs can be implemented on data-flow architectures, but also on Von Neumann architectures without any overhead for a token-based control mechanism.* To achieve this, SIGNAL handles a quite new notion of time, which makes this language very close to real-time ones, as the reader may convince himself.

1.1 The notion of time in SIGNAL: synchronous languages.

Real-time synchronous languages refer to the notion of time in a completely new point of view. This point of view was first taken by the imperative language ESTEREL [Berry & Cosserat 1984, Tanzi 1985], and is also used by the data-flow oriented languages LUSTRE [Bergerand & al. 1985] and SIGNAL. Let us outline the principles of synchrony. Synchronous real time systems differ from asynchronous ones in the two following aspects:

- *concerning the internal mechanisms of the system:* every action (communication or operation on data) is instantaneous, i.e. has zero duration;
- *concerning the communications with the external world:* the set of the possible input stimuli is fixed and known in advance, and input flows are specified through both 1/ the values they carry, and 2/ a total ordering of the «instants» at which these values are available at the input ports.

We refer the interested reader to [Berry & Cosserat 1984] for a detailed discussion about the relevance of the assumption of zero duration of the actions. Of course, the assumption 2/ above is the fundamental feature which characterizes the way synchronous systems do communicate with the external world, compared to asynchronous ones. Let us illustrate further this point on a simple example.

Consider a real time system with two inputs

1. a data input carrying an ordered file of data, named x;

2. an interrupt input port named s.

Then, the specification of an input history according to the synchronous point of view must be of the form

$$\begin{array}{cccc} \perp & s_2 & \perp & s_4 \ldots \\ x_1 & x_2 & x_3 & \perp \ldots \end{array}$$

i.e. both the values (as usually, $\perp$ denotes the absence of data) and their interleaving must be specified. The time reference is nothing but the index $t = 1,2,3,..$ of the successive flashes of data; this notion of time is thus *local*, i.e. assigned to a given system. When several subprocesses communicate, their own time is generally subordinated to the time of the global system, this point will be clarified in the sequel. To summarize, *the essentially nondeterministic character of the asynchronous communications with the external world is concentrated here in some (ignored) external mechanism which* decides *this global ordering.* Primitive required to specify and implement these mechanisms are precisely the only ones that are missing to let SIGNAL be a real-time language in the usual sense.

A key feature which characterizes the language SIGNAL with respect to the other synchronous languages such as ESTEREL and LUSTRE is its ability to run according to a mixed *passive* (i.e. data-driven in the data-flow framework) and *active* (i.e. demand-driven) mode of communication with the external world. On the other hand, for example, ESTEREL possesses only the passive mode. This basic feature allows in SIGNAL to tune internal clocks to the external communications, that are *more frequent* than the clock of the input stimuli. This mechanism is fundamental in both C^3-systems and in RTL-level simulations of SIGNAL programs.

1.2 About the semantics of SIGNAL and its clock calculus.

Apart from the CSP denotational model of the language OCCAM [Brookes & al. 1984], we must emphasize on the denotational model of Dynamic Network Processes introduced by Kahn [Kahn 1974], and further studied by De Bruin and Boehm [De Bruin & Boehm 1985]. This model reveals the difficulty to handle the denotational approach in data-flow oriented systems transforming histories, which is the case of SIGNAL; in fact the second author recognized the same difficulty when writing a simple denotational model for synchronous systems [Benveniste 1985]. The reason is that to study the causality correctness of networks of processes requires the use of a difficult *continuation* technique [De Bruin & Boehm 1985] to prove the existence of correct fixpoint

solutions to the corresponding system of equations in the semantic domain. On the other hand, the reader can also refer to [Berry & Cosserat 1984] for an interesting discussion about the use of algebras of processes such as CCS [Milner 1980] or MEIJE [Boudol & al. 1984] as a semantic domain for the language ESTEREL. We choosed to follow the same approach as [Berry & Cosserat 1984] to give the semantics of SIGNAL, i.e. to use a *direct structural operational semantics* a la Plotkin [Plotkin 1981].

The larger flexibility of SIGNAL possesses also its drawbacks as couterparts, let us discuss briefly this point. First, some elementary SIGNAL processes induces synchrony constraints at their input ports; for example x := a+b requires that the inputs a and b be simultaneously available, i.e. have the same rate; this causes a great advantage in the efficiency of the implementation of SIGNAL programs, but this turns out to allow the programmer to write programs that can be time-incorrect. Second, the counterpart of this flexibility is that a SIGNAL process is generally nondeterministic, even if correct. The kind of semantics we use allows us to clarify these points, and to give precise definitions for «time-correctness», «determinism» and so far. As a matter of fact, this approach provides us with a simple mechanism (the technique of «cuts») *to reduce a SIGNAL process to some system of polynomial equations over the commutative field Z/3Z which summarizes the timing behaviour of this process: its CLOCK CALCULUS.* The checking of all timing properties of a process (such as time-correctness, determinism,...) can be directly read on the solution of this system of polynomial equations; let us point out that all polynomial functions in *Z/3Z* are of degree at most 2, so that the algorithm is rather staightforward.

A byproduct of the clock calculus is the possibility to get necessary and sufficient conditions for a SIGNAL process to exhibit deadlocks. The key tool is the *conditional dependence graph*, where the data dependencies are labelled by the clock that causes the considered dependency. This allows SIGNAL to accept as correct programs that are currently rejected by the dependency checker of ESTEREL. Finally, this conditional dependence graph is the convenient intermediate level of compilation for SIGNAL programs which allows *separate compilation.*

Finally, we should emphasize that we do not use the formalism of transitions as a part of the compiler of SIGNAL. Transitions are only used as theoretical tools to study notions and prove theorems. The only objects the compiler will handle are the clock calculus and the conditional dependence graph, as the computational semantics shows. The advantage is that these objects can be handled far more efficiently than transitions, so that we hope to design an efficient compiler.

1.3 Organization of the article, a guide to the reader.

In the chapter 2 the kernel of SIGNAL is presented as an algebra of processes characterized by sets of transitions. First, elementary processes (called generators) are presented, then the interconnection operators are presented, which generate the complete SIGNAL algebra from these generators. This chapter is thus concerned only with syntax. The basic method is to extend a given functional instantaneous (i.e. classical) language to a language handling flows. This approach allows to introduce the key notion of *computable transitions*, that will be of importance in the sequel.

The behavioral semantics is introduced in the chapter 3; this is the part of the semantics which defines completely the temporal behavior of SIGNAL processes. Fundamental notions such as time-correctness and determinism are formally introduced and illustrated on process transitions.

A sufficient condition for checking the process congruence is given in the chapter 4; this conditions relies on the investigation of some suitable canonical form, which turns out to completely describe the network corresponding to a given process.

The chapter 5 is the fundamental one: the clock calculus is introduced in some informal way, and analysed. Its power is then illustrated on several intricate examples.

The data dependencies are analysed in the chapter 6. The key tool we introduce for this purpose is the conditional dependence graph.

The chapter 7 presents the main theorem, which states tight sufficient conditions to check time-correctness, determinism, and computability (i.e. absence of deadlock due to data dependencies). The powerful method of «cuts» is introduced for this purpose, and provides us with a rigorous justification of the clock calculus, as well as ways to build other clock calculi.

The computational semantics, i.e. the execution scheme of correct SIGNAL processes, is presented in the chapter 8. We show first how to derive a data-flow execution scheme for a correct SIGNAL process; the key point here is that, thanks to the static analysis performed by the clock calculus and the conditional dependence graph, there is no need for multiple token files in the execution of correct SIGNAL programs. Then we give the sequential computational semantics in the sense of [Berry & Cosserat 1984], i.e. we give an algorithm to build an automaton which successively fires all the generators involved in a given event of a SIGNAL process; this automaton is directly derived from the clock calculus and the conditional dependence graph.

Finally, a more subtle example is analysed in the chapter 9: the time-multiplexer. Time-

multiplexing data is a wellknown task, but is impossible to perform with other synchronous languages, since an internal clock must be introduced, which is faster than the clock of the input signal. Nevertheless, the tools we have developped can prove that the proposed SIGNAL program is time-correct and deterministic. Finally, we illustrate on this example the relevance or more powerful clock calculi which could also be derived from the method of cuts.

For a first reading, the reader could get rapidly the flavour of the language and its semantics by reading the chapters 2 (introduction of the language), 5 (clock calculus), and 6 (conditional dependence graph), together with the examples of the chapter 3. These chapters are easy to follow, and provide a good insight on the techniques we used. For example, the clock calculus itself is a good way to get intuition about time-correctness.

Chapter Two

THE KERNEL OF THE LANGUAGE SIGNAL

In this chapter, we introduce the kernel language SIGNAL, which we shall call simply SIGNAL, using the syntax of the transitions rules a la Plotkin. The principle of the kernel of the language SIGNAL is the following: we consider that an instantaneous functional language is given. Here, "instantaneous" means that the objects these functions are applied to are no more flows of data, but rather single elements. That is to say we assume we know the notions of *functions* and their domains of definition, and that we know a theory to calculate the dependencies induced by these functions. Then, the only job we shall accomplish is to extend this given instantaneous language to a new one, where the functions are no more applied to single elements, but rather transform histories into histories; this claim has to be understood in an informal fashion, since this point will in fact only be investigated in the *behavioral semantics* we shall give in the next chapter.

2.1 Notations, definitions, and axioms

2.1.1 The given instantaneous language

For the kernel language, we shall consider that a standard theory of functions is available; furthermore, recursivity will be forbidden. The basic instructions of this language are

$$X := \text{exp} \tag{2-1}$$

where *exp* denotes generically a set of expressions known by this language; here, X denotes a formal value, while *exp* generally involves other formal values. A program is nothing but a set of such assignments, where identical names refer to identical formal values. Among the set of expressions, we shall distinguish the *primitive boolean expressions*, such as

$$x < y$$

whereas *boolean expressions* are built from primitive ones using the boolean operations *and, or, not.*

The only important notion we shall use is the notion of *dependency analysis* (or *causality* in the

framework of [Berry and Cosserat 1984]) of a system of functions. To deal with this problem, we shall introduce the notation

$$X(a,b,c) \tag{2-2}$$

to express that the formal values a,b,c appear as arguments in the right handside of the assignment $X:=\exp$. This relation is extended to be transitive. In our instantaneous language, recursivity will be forbidden; programs that do not exhibit recursivity will be called *computable*. The following criterion will be used in the sequel:

DEFINITION 1 : An instantaneous program is computable if and only if there is no formal value x such that

$$x(x) \quad \square \tag{2-3}$$

A computable program thus defines a set of functions, together with their compositions rules through the identity of the names of some of their arguments. As a consequence, in a computable program, suitable substitutions yield an equivalent program where *every formal value X depends only upon free formal values*. This finishes the presentation of the instantaneous language.

2.1.2 SIGNAL expressions.

To describe a SIGNAL expression, we shall use the formalism of transition rules a la Plotkin. The following example shows such a syntax:

$$< mem[omem] > \quad \frac{p[x]}{q[y:=\exp]} \rightarrow \quad < mem[nmem:=\exp'] > \tag{2-4}$$

In this example, *mem* denotes a memory, *p* an input port, and *q* an output port; *mem[omem], p[x]* expresse the fact that *mem* and *p* respectively carry the values *omem* and *x*, whereas the instructions $y:=\exp$, $nmem:=\exp'$ have to be taken in the sense of the instantaneous language we have introduced before.

We shall need a special value, called *undefined*, and denoted for short by $\perp$. This value does not

belong to the domains of the formal values of the instantaneous language we have introduced. Roughly speaking, $\perp$ has to be interpreted as the absence of value.

We are now ready to introduce SIGNAL expressions.

DEFINITION 1 : a SIGNAL expression is a 5-uple

$$\mathbf{P} \ \{ \ \$P \ ?P \ !P \ \} = TRANS$$

where

- **P** is the name of the expression
- *$P* is a finite set of memories (or states)
- *?P* is a finite set of input ports
- *!P* is a finite set of output ports
- *TRANS* is a finite set of transitions (see below) .

Memories and ports will generically be referred to as *carriers*. A *transition* is a rule of the form

$$< \$P > \ \frac{?P}{!P} \rightarrow \ < \$P' > \qquad (2\text{–}5)$$

where

- *$P* denotes a list of terms of the form *z*[*omem*] where *z* is a memory and *omem* the value carried by *z* before the transition;
- *$P'* denotes a list of terms of the form *z*[*nmem:* = exp] , where the memories are the same as before, *nmem* is the value carried after the transition, and *nmem:* = exp is an instruction of the instantaneous language;
- *?P* is a list of terms of the form *p*[*x*], where *p* is an input port carrying the value *x*, or of the form *p*[$\perp$]; if *p* is of boolean type, the expression *p*[*x*]; *x* = *tt*, *ff* expresses that the considered transition can be applied when the carried value *x* is respectively true or false;
- *!P* is a list of terms of the form *q*[*y:* = exp] , where *q* is an output port carrying the value *y*, while *y:* = exp is an instruction of the instantaneous language, or of the form *q*[$\perp$]; the expression *q*[*y:* = exp]; *y* = *tt*, *ff* is used in the same way as before.

The set of the memories, input and output ports is defined in a *static* way, while carried values depend on the transitions as we shall see later. ■

Example

$$P\{\ \$in\ ?in\ !out\} = \ < in[x] > \ \frac{in[y]}{out[z:=\ if y > 0\ \ then\ x + y\ else\ x]} \rightarrow \ < in[x:=y] >$$

denotes the program " out_t = if $in_t > 0$ then $in_{t-1} + in_t$ else in_{t-1} ". We are now ready to state the axioms that a SIGNAL process must satisfy.

2.1.3 Axioms

Every SIGNAL expression P must satisfy the following list of axioms:

S-AXIOMS :

- S1: different input ports must have different names; different output ports must have different names (however a common name can be used for an input and an output port of the same process).
- S2: if every port of a given transition carries the value ⊥ , then the values in the memories are unchanged after the transition. □

COMMENTS: The axiom S1 means that ports are labelled via names. The axiom S2 is rather fundamental: it expresses that nontrivial transitions require communications with the external world; of course, only nontrivial transitions have to be given to specify a process.

We are now ready to introduce the basic instructions of SIGNAL using the syntax of the transition rules.

2.2 The basic instructions

Here follow a list of the basic instructions of the language SIGNAL, and the corresponding shortened syntax we shall use to refer to them:

generators (names refer to ports):

- *function*: $(q_1, \ldots, q_l) := \exp(p_1, \ldots, p_k)$
- *delay*: $q := \$p$
- *filter*: $out := in$ **when** $control$
- *merge*: $out := main$ **default** $second$
- *condition*: $h := \mathrm{tt}(C)$

connection operators (capitals refer to processes, whereas lower cases refer to ports):

- *relabelling of input ports*: $Q = P?a:b$
- *relabelling of output ports*: $Q = P!a:b$
- *collateral*: $P = Q\&R$
- *p-connection*: $Q = P@x$

2.2.1 Generators

2.2.1.1 Functions

Let

$$y_1 := \exp_1$$
$$\ldots$$
$$y_k := \exp_k$$

be instructions of the static language, where, according to (2-2),

$$y_1(x_1, \ldots, x_n)$$
$$\ldots$$
$$y_k(x_1, \ldots, x_n)$$

and set $\exp = (\exp_1, \ldots, \exp_k)$; then

$$(q_1, \ldots, q_k) := \exp\ \{\$.\ ?p_1, \ldots, p_n\ !q_1, \ldots, q_k\}$$

$$<.> \frac{p_1[x_1]\ldots p_n[x_n]}{q_1[y_1 := \exp_1]\ldots q_k[y_k := \exp_k]} \rightarrow <.> \qquad (2\text{-}6)$$

COMMENTS: The condition that all input values must be defined plays a fundamental role in SIGNAL: it means that all input ports of a function must be involved in a non trivial transition . For example, we refuse to assign any meaning to $x := y+z$ when y and z are not simultaneously available; as a matter of fact, we assume that such event should never occur if the program were correct . Recall that this is an arbitrary choice which is consistent with the domain of application we have in mind. This instruction extends the classical static functions to functions acting on flows. For instance, $x := y+z$ roughly means $x_t := y_t + z_t$ for every instant t, although this interpretation has to be handled with some care, since our notion of time is not unique, but rather multiform, as we have explained before.

2.2.1.2 Delay

$$out := \$in \ \{\$in \ ?in \ !out\}$$

$$=$$

$$<in[x]> \frac{in[y]}{out[w := x]} \rightarrow <in[x := y]> \qquad (2\text{-}7)$$

COMMENT: the delay behaves then like a fifo register.

2.2.1.3 Filter

$$out := main \ \textbf{when} \ control \ \{\$. \ ?main, control \ !out\}$$

$$=$$

$$<.> \frac{main[x]\,control[y]}{out[w := x]} \rightarrow <.>$$

$$<.> \frac{main[\perp]\,control[y]}{out[\perp]} \rightarrow <.>$$

$$<.> \frac{main[x]\,control[\perp]}{out[\perp]} \rightarrow <.> \qquad (2\text{-}8)$$

COMMENT: the control port acts as a control signal: the value carried by the main port is lost if this control signal fails to be available.

2.2.1.4 Merge

$$out := main \ \textbf{default} \ second \quad \{\$.\ ?main, second\ !out\}$$

$$=$$

$$<.> \frac{main[x]\,second[y]}{out[w := x]} \rightarrow <.>$$

$$<.> \frac{main[x]\,second[\perp]}{out[w := x]} \rightarrow <.>$$

$$<.> \frac{main[\perp]\,second[y]}{out[w := y]} \rightarrow <.> \qquad (2\text{-}9)$$

COMMENT: Please, note that this "merge" operator is deterministic, since a priority has been stated in his definition.

2.2.1.5 Condition

Here, c denotes a port of boolean type:

$$h := \mathrm{tt}(c) \quad \{\$.\ ?c\ !h\}$$

$$=$$

$$<.> \frac{c[z];\ z = tt}{h[w := tt]} \rightarrow <.>$$

$$<.> \frac{c[z];\ z = ff}{h[\perp]} \rightarrow <.> \qquad (2\text{-}10)$$

where *tt, ff* respectively denote the boolean values *true, false*. In other words, the *condition* extracts the instants at which a boolean value is true.

CONCLUSION: It is easy to verify that these generators are SIGNAL processes, i.e. verify the S-AXIOMS. S1 has to be verified on the syntax of each instance of the given generator; on the other hand, the other axioms are trivially satisfied.

SHORTENED NOTATION: To avoid to write several times trivial assignments such as p[x := u] , we shall simply substitute the corresponding formal values, thus writing for instance p[u], and substituting x by u in the considered transition. For instance, the first transition of when will be written simply

$$<.> \frac{main[x] \;\; control[y]}{out[x]} \rightarrow <.>$$

This shortened notation will be used in the sequel.

2.2.2 Interconnection operators

The presentation of these follows the *structural conditional rewriting rules* of Plotkin, we shall comment on the first instruction. In the sequel, $, $' denote for short the list of the memories of a process, together with the values they carry; *?list* (resp. *!list*) denotes a list of input (resp. output) ports together with the values they carry. In the sequel, the notation « P: transition » will be used to indicate that the considered transition is a transition of the process P.

2.2.2.1 Relabelling of input ports

Q = P?a:b is obtained as follows when both *a* and *b* are input ports of P: the same values are broadcasted to the ports *a* and *b* , and this results in a single port, named *b* . This is the only case in which the relabelling is different from a crude change of names.

(i) *a* doesn't belong to *?P* :

$$P?a:b = P$$

(ii) *a* belongs to *?P* but *b* doesn't:

$$\frac{P:\ <\$> \ \xrightarrow[!list']{a[x]\,?list}\ <\$'>}{P?a{:}b\ :\ <\$> \ \xrightarrow[list']{b[x]\,?list}\ <\$'>}$$

with the corresponding transition with ⊥ instead of x.

(iii) *a* and *b* belong to *?P* :

$$\frac{P:\ <\$> \ \xrightarrow[!list']{a[x]\,b[x]\,?list}\ <\$'>}{P?a{:}b\ :\ <\$> \ \xrightarrow[!list']{b[x]\,?list}\ <\$'>}$$

with the corresponding transition with ⊥ instead of *x*.

HOW TO READ THE RULES: these rules are presented according to the syntax

$$\frac{P:\ transition1}{Q:\ transition2}$$

The rule means that, to know the transitions of Q, one has to substitute to the transition1 of P the corresponding transition2. Note that, in the transition (iii), the two input values carried by *a* and *b* are constrained to be the same.

PROPERTIES OF INPUT RELABELLING:

(a) If *P* satisfies the axioms S1 and S2, so does *P?a:b*

(b) $?(P?a{:}b) = (?P \cup \{b\}) - \{a\}$ if $a \in ?P$

2.2.2.2 Relabelling of output ports

To satisfy the axiom S1, the relabelling *Q = P!a:b* is not defined when both *a* and *b* are output ports of *P*. Otherwise, the meaning of this operator is obvious, and its definition is as follows.

(i) *a* doesn't belong to *!P* :

$$P/a{:}b = P$$

(ii) a belongs to $/P$:

$$\frac{P: \ <\$> \xrightarrow[/list'\ a[x]]{?list} <\$'>}{P/a{:}b: \ <\$> \xrightarrow[/list'\ b[x]]{?list} <\$'>}$$

with the corresponding transition with $\perp$ instead of x.

PROPERTIES OF OUTPUT RELABELLING:

(a) if P satisfies the axioms S1 and S2, and if a and b are not both output ports of P, then $P/a{:}b$ satisfies also these axioms.

(b) $/(P/a{:}b) = (/P \cup \{b\}) - \{a\}$ if $a \in /P$.

2.2.2.3 Collateral

The operator collateral, denoted by $Q = P_1 \ \& \ P_2$ is defined only when P_1 and P_2 have no common output port, to satisfy the axiom S1. The same values are broadcasted to the input ports having the same name. When P_1 and P_2 have no common input port, these processes are allowed to participate or not to a transition of the resulting process Q. The definition of the collateral is now given. In this definition, $list[\perp]$ means that the mentioned ports all carry the value $\perp$; this notation will be useful to mention that one of the subprocesses building the collateral does not participate to the considered transition of the resulting process Q. The label $\$$ refers to a list of memories, together with the values they carry.

(i) P_1 and P_2 have no common input port:

(i.a) P_2 doesn't participate to the transition:

$$\frac{P_1: \ <\$_1> \xrightarrow[/list'_1]{?list_1} <\$'_1> \qquad P_2: \ <\$_2> \xrightarrow[/list'_2[\perp]]{?list_2[\perp]} <\$_2>}{P_1\&P_2: \ <\$_1 + \$_2> \xrightarrow[/list'_1;\ /list'_2[\perp]]{?list_1;\ ?list_2[\perp]} <\$'_1 + \$_2>}$$

(i.b) P_1 doesn't participate to the transition: symmetric definition

(i.c) P_1 and P_2 both participate to the transition:

$$\frac{P_1:\ <\$_1> \xrightarrow[!list'_1]{?list_1} <\$'_1> \qquad P_2:\ <\$_2> \xrightarrow[!list'_2]{?list_2} <\$'_2>}{P_1\&P_2:\ <\$_1 + \$_2> \xrightarrow[!list'_1;\ list'_2]{?list_1;\ ?list_2} <\$'_1 + \$'_2>}$$

(ii) If a belongs to $?P_1$ and $?P_2$, then choose a name b which is not a member of $?P_1 \cup ?P_2$; the formula

$$P_1\&P_2 = ((P_1 ?a{:}b)\ \&\ P_2)\ ?b{:}a$$

allows to define P_1 & P_2 by induction.□

COMMENT: $\$_1 + \$_2$ denotes the direct sum of sets. In other words, processes interconnected in collateral do not share variables.

PROPERTIES OF COLLATERAL:

(a) If P_1 and P_2 satisfy the axioms S1 and S2, then so does P_1 & P_2.

(b) The operator & is commutative and associative.

(c) The & satisfies the following properties:

- $?(P\&Q) = ?P \cup ?Q$
- $!(P\&Q) = !P \cup !Q$
- $\$(P\&Q) = \$(P) + \$(Q)$

where $\$(P)$ denotes the set of the memories of P .□

PROOF: rather easy although tedious.

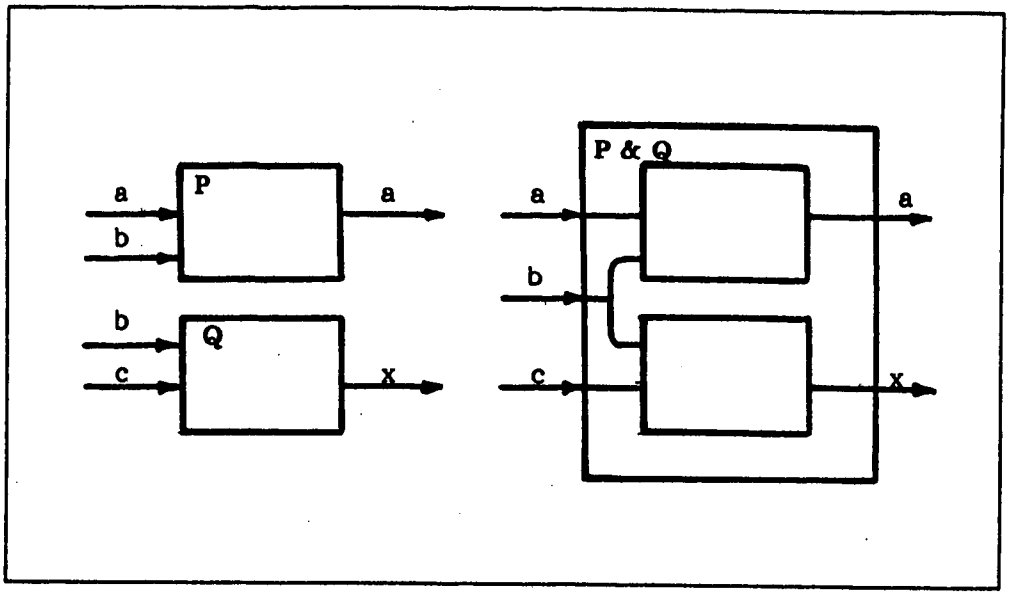

Figure 1. Block-diagram of the collateral

2.2.2.4 The p-connection

The p-connection, denoted by $Q = P@b$, is intended to connect the output port named b to the input port of P possessing the same name. This operator is the key of process interconnections.

(i) if b doesn't belong to both $?P$ and $!P$, then

$$P@b = P.$$

(ii) Otherwise

$$\frac{P: \quad <\$> \xrightarrow[!list' \ b[x]]{?list \ b[x]} <\$'>}{P@b: \quad <\$> \xrightarrow[!list' \ b[x]]{?list} <\$'>} \qquad (2\text{-}11)$$

with the corresponding transition with $\perp$ instead of x.

WARNING: this rule is in general not effective. In fact, while values carried by input ports are free (since they are formal values), values carried by output ports are not! As a consequence, we don't

know in general the meaning of the requested transition of the original expression P, where the value x is requested at both the input port b and the output port b. To study this problem, let us investigate in more details the possible transitions of the original expression P; according to (2-5), a transition of P is of the form

$$<\$> \frac{?list\ b[y]}{!list'\ b[x := \text{exp}]} \rightarrow <\$'>$$

where $\$$, *list*, are the usual shortages. Then, two situations may occur

- $x(y)$ holds, so that the requested transition (2-11) for P would result in the forbidden dependency $x(x)$;
- $x(y)$ does not hold, so that we are free to replace the formal value y carried by b by the formal value x, so that the requested transition (2-11) clearly makes sense in this case. We shall say in this case that this transition is *computable*. SIGNAL expressions all transitions of which are computable are said to be *computable*.

If (2-11) is computable for every transition of P, we shall say that *b is free in P*. Then, the following theorem is easy to prove:

PROPERTIES OF p-CONNECTION:

(a) If P satisfies S1 and S2, then $P@b$ can be defined if and only if b is free in P; in this case, $P@b$ also satisfies S1 and S2.

(b) In this case, we have, if $b \in ?P \cap !P$

- $?(P@b) = ?P - \{b\}$
- $!(P@b) = !P$ □

In the sequel, we shall provide an algorithm to detect whether or not a requested p-connection is feasible; the job we have to achieve is to detect instantaneous short-circuits in the dependencies of the data at a given instant.

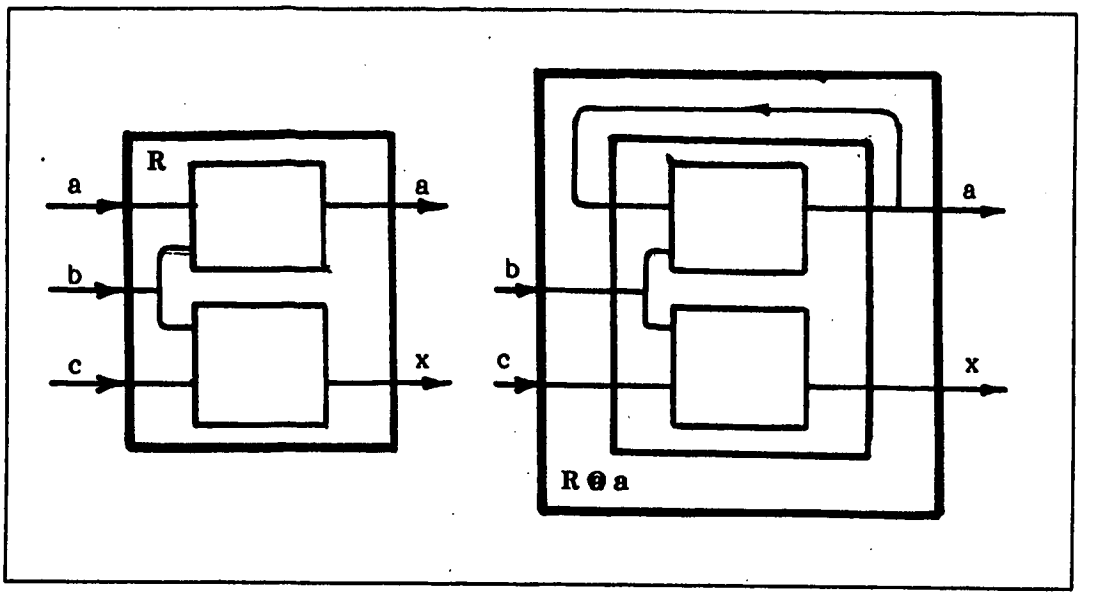

Figure 2. Block-diagram of the p-connection

CONCLUSION : We have introduced the language SIGNAL using the syntax of Conditional Rewriting Rules of Plotkin. The generators and interconnection operators generate an algebra of SIGNAL expressions: members of this algebra will be called *SIGNAL processes*, or *processes* for short.

Chapter Three
THE BEHAVIORAL SEMANTICS.

In the preceding chapter, we have defined SIGNAL processes as sets of transitions. The purpose of the behavioral semantics is to define how SIGNAL processes transform histories. Before to present this semantics, we shall need some further notations and definitions.

3.1 Notations and definitions.

We shall denote by small greek letters an arbitrary effective value associated with a formal one. For example, if n is a formal value of integer type, v will denote one among the integers 1,2,3,... Effective values can be substituted for formal values in instructions of the static language, thus returning another effective value as a result.

Given a port p of a SIGNAL process, we shall denote by $V(p)$ the domain of the effective values carried by p, and we set

$$V^{*}(p) = V(p) \cup \{\perp\} \tag{3-1}$$

3.1.1 Events and histories.

3.1.1.1 Input events.

Let P be a SIGNAL process, and let $?P = \{p_1, \ldots, p_n\}$ be the set of the input ports of P. An *input event* of P is an n-uple

$$\varepsilon = \{(p_1, \varepsilon_1), \ldots, (p_n, \varepsilon_n)\}$$

where

$$\{\varepsilon_1, \ldots, \varepsilon_n\} \in V^{*}(p_1) x \ldots x V^{*}(p_n)$$

3.1.1.2 Input histories.

An *input history* of *P* is a (possibly infinite) sequence $\{\varepsilon(t)\}_{t\geq 0}$ of input events of *P*.

To get insight in the behavioral semantics, the reader should consider that the intuitive meaning of histories is the following: given a process *P*

- the status of its memories before a transition summarizes the *past* of *P*
- the input event $\varepsilon(0)$ is the *present* of *P*
- the other part of the input history, namely $\{\varepsilon(t)\}_{t>0}$ represents the *future* of *P*.

3.1.2 Allowed transitions, accceptances and refusals.

We shall denote by T(P) the set of the transitions of P, and by $o\mu$ the k-uple of the effective values carried by the memories of P before these transitions .

3.1.2.1 Allowed transitions.

Given an input event ε, we shall denote by $T(o\mu, \varepsilon)$ the sets of the non trivial computable[1] transitions of P that *accept* ε in the following sense

$$T \in T(o\mu, \varepsilon)$$

$$\Leftrightarrow$$

$$T = \quad < \$[o\mu] > \frac{p_1(\varepsilon_1) \ldots p_n(\varepsilon_n)}{f} \rightarrow < \$[n\mu] > \qquad (3\text{-}2)$$

$T(o\mu, \varepsilon)$ is the set of the transitions that are *allowed* by $(o\mu, \varepsilon)$. For a process all transitions of which are computable, acceptances can be checked immediately, since the expressions carried by the output ports can be rewritten so as to depend upon input values only.

---

1 see (2-11)

3.1.2.2 Acceptances, refusals.

The pair $(o\mu, \varepsilon)$ is an *acceptance* of P if $T(o\mu, \varepsilon) \neq \phi$; otherwise, $(o\mu, \varepsilon)$ is a *refusal* of P.

3.1.3 Actions and runs.

3.1.3.1 Actions

Given a process P, with $\{\varepsilon(t)\}_{t \geq 0}$ as input history, and denoting by T an element of $T(o\mu, \varepsilon(0))$, an *action* of P is a map

$$(o\mu, \varepsilon(0)) \xrightarrow[f]{} (n\mu, \varepsilon(1))$$

defined by

$$T = \langle \$[o\mu] \rangle \xrightarrow[f]{p_1(\varepsilon_1(0)) \ldots p_n(\varepsilon_n(0))} \langle \$[n\mu] \rangle \qquad (3\text{–}3)$$

Note that a process generally possesses several actions, since $T(o\mu, \varepsilon)$ is generally not a singleton.

3.1.3.2 Runs.

A finite *run* of P is a finite iteration of actions of P:

$$(o\mu(0), \varepsilon(0)) \xrightarrow[f_0]{} (o\mu(1), \varepsilon(1)) \xrightarrow[f_1]{} \ldots \rightarrow (o\mu(t), \varepsilon(t)) \qquad (3\text{–}4)$$

of actions of P. Since the set of finite runs is ordered by prefixing, standard continuity arguments allow to extend this notion to *denumerable runs*. A finite or denumerable run will simply be called a *run*.

3.1.4 Example 1.

We shall illustrate on a simple example the behavior of a SIGNAL process. Consider the following program, where, to simplify, we shall assume that k is a pure clock, i.e. possesses only a single value T:

P {$n ?k !y,z,n} = ((y := z+1) & (z := $n) & (n := y when k)) @ y,z,n

The transitions of the generators are

function +

$$<.> \frac{z[u]}{y[w := u+1]} \rightarrow <.> \qquad (3\text{-}5)$$

delay

$$<n[u]> \frac{n[v]}{z[u]} \rightarrow <n[v]> \qquad (3\text{-}6)$$

when

$$<.> \frac{y[v]\ k[T]}{n[v]} \rightarrow <.> \quad (i)$$

$$<.> \frac{y[\perp]\ k[T]}{n[\perp]} \rightarrow <.> \quad (ii)$$

$$<.> \frac{y[v]\ k[\perp]}{n[\perp]} \rightarrow <.> \quad (iii) \qquad (3\text{-}7)$$

Transfers due to the &'s, and to the @'s yield finally the set of transitions

$$<n[u]> \frac{k[T]}{z[u]\ y[u+1]\ n[u+1]} \rightarrow <n[u+1]> \quad (i)$$

$$< n[u] > \frac{k[T]}{z[\perp]\ y[\perp]\ n[\perp]} \rightarrow\ < n[u] > \quad (ii) \tag{3-8}$$

The transition (3-8-i) results from the selection of the transition (3-7-i) in the application of the rules of the collateral and of the p-connection; the transition (3-8-ii) is due to the selection of the transition (3-7-ii), whereas the selection of the transition (3-7-iii) is forbidden because of the rule of the p-connection. If the initial value of the memory is 0, a provable run is

$$\begin{aligned} < n[0] > & \frac{k[T]}{z[0]\ y[1]\ n[1]} \rightarrow \\ < n[1] > & \frac{k[T]}{z[1]\ y[2]\ n[2]} \rightarrow \\ < n[2] > & \frac{k[T]}{z[2]\ y[3]\ n[3]} \rightarrow \ldots \end{aligned} \tag{3-9}$$

which corresponds to a counter of the occurrences of the input *k*. But, since anyone of the transitions (3-8) can be fired with the same input stimulus, this process is obviously non deterministic: for example, an other provable run is

$$\begin{aligned} < n[0] > & \frac{k[T]}{z[\perp]\ y[\perp]\ n[\perp]} \rightarrow \\ < n[0] > & \frac{k[T]}{z[0]\ y[1]\ n[1]} \rightarrow \\ < n[1] > & \frac{k[T]}{z[\perp]\ y[\perp]\ n[\perp]} \rightarrow\ < n[1] > \ \ldots \end{aligned} \tag{3-10}$$

In fact, this example shows a situation similar to the one studied in the counterexample of [Brock & Ackermann, 1981], and we shall see later that SIGNAL is a convenient answer to the paradox studied therein.

3.2 Some problems.

3.2.1 Computability of a transition.

Recall that a transition is said to be *computable* if the static instructions it uses does not exhibit short circuits such as $x(x)$ in the sense of (2-3). We need effective criteria to recognize computable from non computable transitions, and to prove that all the transitions of a process will be computable. This will be the subject of the chapter about data dependencies.

3.2.2 Time-correctness.

Given a process P and a subset $\{p_1, \dots p_n\}$ of ?P, we shall say that the set $\{p_1, \dots, p_n\}$ is *time-incorrect* if

$$\exists \xi = \{\xi_i\}_{i=1\dots n},\ \xi_i \text{ effective value, such that}$$
$$\forall(\varepsilon, \eta) \text{ input event with } \varepsilon = \{(p_1, \varepsilon_1), \dots, (p_n, \varepsilon_n)\},\ \varepsilon_i = \xi_i \text{ or } \varepsilon_i = \perp,$$
$$\{(o\mu; \varepsilon, \eta) \text{ acceptance of } P\} \Rightarrow \{\varepsilon_i = \perp\ \forall i\}$$

Roughly speaking, *a subset of ports is time-incorrect if there exists a memory status and a set of candidate effective values for these ports which cannot be used whatever the other input candidates are.*

A process which possesses no time-incorrect subset of input ports is said to be *fully time-correct.* Criteria of time-correctness will be obtained thanks to the *clock calculus* we shall introduce later.

3.2.2.1 Example 2

Consider the following program

P { $. ?a,b !z,x } = ((x := a **default** b) & (z := a+x)) @x

The function " + " requires that *a* and *x* be available simultaneously, which implies that *a* must be available every time *b* is. This illustrate the fact that a SIGNAL process can exhibit synchronization constraints at its input ports. This is a fundamental remark, since the environment of this process must be in accordance with such constraints. Note that this process is fully time-correct.

3.2.2.2 Example 3

Consider the following program

P { $. ?a,b !h,x,z } = ((h := tt[a>0]) & (x := a **when** h) & (z := b+x)) @h,x

This program is intuitively time-correct; the relative timing of the input ports a and b is determined by the occurrences *tt* of the condition [a>0], so that the synchronisation constraint on the input ports of P is somewhat more subtle in this case. This is enlightened by the set of the transitions of P, which are obtained using the rules we have defined:

$$<.> \frac{a[u]\ b[v]}{(a>0)[y:=u>0]:y=tt\quad h[tt]\ x[u]\ z[w:=u+v]} \rightarrow <.> \qquad (i)$$

$$<.> \frac{a[u]\ b[\perp]}{(a>0)[y]:y=ff\quad h[\perp]\ x[\perp]\ z[\perp]} \rightarrow <.> \qquad (ii)$$

An acceptance can always be extracted from any candidate input effective values for *a* and *b*, so that this process is fully time-correct; however, note that *b* can starvate because of its environment since it is never used if *a* is always ≤ 0, but it cannot starvate by itself.

3.2.2.3 Example 4

Consider the following program

P { $. ?a,b !h,x } = ((h := tt[a<b]) & (x := a **when** h) & (z := x+b)) @h,x

The meaning of this program is "add *b* to *a* if *a*<*b*". Evaluating the condition [a<b] requires that a and b be available at the same rate. However, the instruction $z := x + b$ requires that *x* and *b* be also available at the same rate. But, because input values are free, the value of [*a* < *b*] must be sometimes *tt*, sometimes *ff*, so that *h* is strictly less frequent than the signals *a* and *b*; but, on the other hand, *x* is at least as frequent as *h*, which is a contradiction. This can be shown on the transitions as follows. The application of the rules shows that P possesses a single transition

$$<.> \frac{a[u]\ b[v]}{(a<b)[z:=u<v]:z=tt\ h[tt]\ x[u]\ z[w:=u+v]} \rightarrow <.> \qquad (3\text{-}11)$$

which can be rewritten as

$$<.> \frac{a[u]\ b[v]\ :\ u<v}{(a<b)[z:=u<v]\ h[tt]\ x[u]\ z[w:=u+v]} \rightarrow <.> \qquad (3\text{-}12)$$

so that no acceptance can be extracted from the input candidate ($u = 1, v = 0$). Consequently, the set of all the input ports of P is time-incorrect.

3.2.3 Determinism.

As usually a process P is said to be *deterministic* if the set of its runs defines a continuous function from the set of the input histories into the set of output histories. A SIGNAL process is generally non deterministic, as the example 1 has shown. One of the main contributions of the clock calculus is to give also simple and effective criteria to check in a static way whether or not a SIGNAL process is indeed deterministic. The example 1 is a non deterministic process, since the set of the allowed transitions is not a singleton. An other example is the following.

3.2.3.1 Example 5

Consider the following program, which is a slight modification of the example 1:

P {$n ?k !y,z,n,x} = ((y := z+1) & (z := $n) & (n := y **when** x) & (x := k **default** z) @ y,z,n,x

The application of the rules yields the following transitions for this process:

$$<n[u]> \frac{k[T]}{z[u]\ y[u+1]\ n[u+1]\ x[T]} \rightarrow <n[u+1]> \quad (i)$$

$$<n[u]> \frac{k[T]}{z[\perp]\ y[\perp]\ n[\perp]\ x[T]} \rightarrow <n[u]> \quad (ii)$$

$$< n[u] > \frac{k[\perp]}{z[u]\ y[u+1]\ n[u+1]\ x[u]} \rightarrow < n[u+1] > \quad (iii) \qquad (3\text{-}13)$$

These transitions accept the same input event with the same memory status: this process is non deterministic. As a matter of fact, if the transition (iii) is always selected, the process never uses the input event. Note that in the examples 1 and 5, if the locations of the $\perp$ at the output ports are specified together with the memory status and input event, then only a single transition can be chosen. Since this property can also be of interest, we shall introduce it now.

3.2.4 Weak determinism.

3.2.4.1 Input-output events.

Let P be a SIGNAL process, and denote by $?P = \{p_1, ..., p_n\}$, $!P = \{q_1, ..., q_k\}$ the sets of the input and output ports of P. an *input-output event* (*i-o event* for short) is a pair

$$(\varepsilon, \kappa),\ \varepsilon \text{ input event},\ \kappa \subset !P$$

As before, i-o event allow us to define *i-o histories.*

3.2.4.2 Input-output-allowed transitions.

Given an i-o event (ε, κ) and a memory status $o\mu$, we shall denote by $T_{i-o}(o\mu, \varepsilon, \kappa)$ the set of the computable transitions of P which are as in (3-2) and furthermore *exhibit* $\perp$ *as carried exactly by those ports belonging to* κ. This set is the set of the i-o-allowed transitions associated to $(o\mu, \varepsilon, \kappa)$.

3.2.4.3 Weak determinism.

A process P is said to be *weakly deterministic* if the set of its runs defines a continuous function from the set of the i-o histories into the set of the output histories. The examples 1 and 5 are weakly deterministic processes, but processes that are not weakly deterministic can be built as well.

CONCLUSION: We have introduced the behavioral semantics of SIGNAL processes. As we have shown, SIGNAL processes are generally non deterministic. Moreover, as we have shown,

the rule of the functions can be a cause of existence of refusals for a given process. Finally, non computable processes can be constructed with the language SIGNAL. As the reader have seen, these properties could be checked on the transitions of the process, although the arguments we have used in the analysis of the example 4 are not obvious to formalize. But this would be a formidable task in the case of complex programs, where the complete set of allowed transitions is long to obtain. The major task of the forthcoming chapters will be to give much more efficient criteria to check these properties in a static way.

Chapter Four
CONGRUENT PROCESSES.

The aim of this section is to give an algorithm to detect if two apparently different processes are in fact identical, in the sense that they behave exactly in the same fashion when connected to any environment. The algorithm simply consists in a reduction of the process to some canonical form, which characterizes any congruence class. The present algorithm is borrowed from [Gautier 1984].

DEFINITION 2 :

(i) Two processes *P* and *Q* are said to be *congruent* if they admit the same transitions, up to a global bijection between the names of their respective memories, and a global bijection between the names of the the carried values of *P* and the carried values of *Q*.

(ii) Two processes *P* and *Q* are said to be *equivalent* if they admit the same transitions up to bijections, respectively between the names of their ports, the names of their memories, and the names of their carried values. □

Our definition of congruence is in accordance with the usual one: two processes are congruent in the sense of the above definition, if and only if they behave similarily when they are connected to a given arbitrary process. On the other hand, it is immediate to transform equivalent processes into congruent ones through a relabelling of their ports, so that we shall only give an algorithm to transform any process into an equivalent canonical form. This section is devoted to the proof of the basic theorem 2.

Before to state and prove this theorem, we shall need some further properties of the interconnection operators. Here, "internal properties" refers to properties involving a single operator, while "external properties" refers to properties involving different operators.

INTERNAL PROPERTIES OF INTERCONNECTION OPERATORS.

1. *&* is associative and commutative.
2. *@* is commutative in the sense that, when *a* is free in *P* and *b* is free in *P@a*, then *b* is free in *P* and *a* is free in *P@b*, and vice-versa. In this case, it is true that *(P@a)@b* = *(P@b)@a*, denoted by *P@a@b*.

PROOF: we shall only prove (2), since (1) is already known. To prove (2), we shall make use of the theory of dependency for static functions. To denote that a static function *x(...)* does not depend on the value *y*, we shall write

$$x(>y<) \tag{4-1}$$

To denote that a value z is substituted for y in a static function x, we shall write

$$x^{y \leftarrow z} \tag{4-2}$$

When z is itself the value of a static function, the substitution results in the composition of the corresponding maps. We are now ready to begin the proof. Let us denote respectively by y and x (resp. y' and x') the values carried by the ports *?a* and *!a* (resp. *?b* and *!b*). Assume that a is free in P, and b free in $P@a$. Then, according to the notation (4-1), we have first

$$x(>y<) \tag{4-3}$$

Thanks to (4-3), it is possible to define $P@a$. According to the rule of the p-connection, the ports *?b* and *!b* of this latter process respectively carry the values

$$y' \quad and \quad x'^{y \leftarrow x}$$

Since b is free in $P@a$, we have

$$x'^{y \leftarrow x}(>y'<) \tag{4-4}$$

But (4-4) implies that

$$x'(>y'<) \tag{4-5}$$

and also

$$x'(>y<) \quad or \quad x(>y'<) \tag{4-6}$$

But (4–5) means that b is free in P, so that we can define $P@b$; as a consequence, the ports $?a$ and $!a$ respectively carry the values

$$y \quad and \quad x^{y' \leftarrow x'}$$

But, thanks to (4–6), we have

$$x^{y' \leftarrow x'}(>y<)$$

so that a is free in $P@b$. That $(P@a)@b = (P@b)@a$ can be proven in the same way. This finishes the proof. □

As a consequence, the above mentioned operators can be used as disordered lists. Note that the relabelling is *non commutative* as yhe following example shows: $(P?a{:}b)?b{:}c$ is generally different from $(P?b{:}c)?a{:}b$, so that, in the sequel, a repeated use of relabellings has to be considered as an ordered list. We are now ready to state and to prove the main theorem of this section.

THEOREM 2 : Every process P is equivalent to a process P_o, which is said to be in *canonical form*, i.e. of the form

$$P_o = (\ \&(\ G\ ?a{:}x\ !b{:}y)\ )\ @c \tag{4–7}$$

where

- G is a generic name of generator
- $?a{:}x$ denotes a list of input relabellings, and $!b{:}y$ a list of output relabellings; a and b denote generic names of ports of generators, while x and y are not port names of generators;
- $\&(...)$ denotes a list of collaterals with (...) as generic subprocess;
- $@c$ denotes a list of p-connections.

P equivalent to P_o will be denoted by

$$P \approx P_o$$

Moreover, this canonical form is unique in the following sense: if P_o and P'_o are two equivalent canonical forms, then there exists a bijection mapping the port names of P'_o on the port names

of P_0 such that the resulting (non ordered) lists $\&(G...)$ and $\&(G'...)$;be the same, up to a permutation on the relabellings .□

PROOF: The formula (4–7) exhibits the following hierarchy between the different interconnection operators:

$$?:, !: < \& < @ \qquad (4\text{–}8)$$

The proof of the theorem (which is a constructive one) relies on the construction of a suitable permutation of the different operators to achieve the ordering indicated in (4–8). Since every permutation is the composition of a sequence of *transpositions* (i.e. permutations involving only 2 terms), the proof of the theorem is a direct consequence of the following list of properties we shall state without proof. In these formulas, by convention, the name *w*, which will appear on the right handside of the character ≈, will generically denote a name of port which is not used in the left handside. Of course, the below listed equivalences are valid only when both sides are well defined SIGNAL processes (i.e. do satisfy the S-axioms). Here follow the list of formulas:

EXTERNAL PROPERTIES OF THE INTERCONNECTION OPERATORS.

- $(P@x)\ \&\ Q \approx (\ (P\ ?x{:}w\ !x{:}w)\ \&\ Q)\ @w$
- $(P@x)\ ?a{:}b \approx (P\ ?x{:}w\ ?a{:}b\ !x{:}w)\ @w$
- $(P@x)\ !a{:}b \approx P@x$
- $(P\&Q)\ ?a{:}b \approx (P?a{:}b)\ \&\ (Q?a{:}b)$
- $(P\&Q)\ !a{:}b \approx P\&Q$ □

The successive use of these formulas provides an algorithm to transform any process into an equivalent process in canonical form. This proves the existence of the process P_0. To prove the unicity of the canonical form, we shall assume that different names of generators do refer to different processes. The proof is then easy, although tedious, if one remarks that the formula

$$P\ ?a{:}x\ ?b{:}y = P\ ?b{:}y\ ?a{:}x$$

holds (as well as the corresponding one for the outputs) provided that *x* and *y* are not input (resp. output) ports of *P* (i.e. the relabelling is commutative in this case); note that this is exactly the constraint we have imposed on the relabellings in a canonical form. This finishes the (sketchy) proof of the theorem 2. □

Chapter Five
THE CLOCK CALCULUS.

This section is the core of the semantics of the language: we shall see that it is indeed possible to check *in a static way* i.e. before the execution of the program, the relative timing of the various events of a process. To do this, we shall first introduce the notion of *clock*. To reason about clocks, we shall restrict the language SIGNAL to a sublanguage called SIG, and map any SIGNAL process into a SIG process. This SIG process will capture and express in a static way the timing of the original SIGNAL process, and its analysis will be performed through the *clock calculus*. It is not the purpose of this chapter to be fully mathematically sounded: we shall rather often refer to the intuition of the reader, and illustrate our purpose through examples. Formal proofs will be given in the chapter 7, where the main theorems are stated and proved, that fully justify the clock calculus.

5.1 Analysis of further simple examples; some consequences.

5.1.1 The example 6

Consider the following program:

P {$x ?a !h,x,zx} = (h := tt[zx>0]) & (zx := $x) & (x := a when h) @ x,zx,C

This program exhibits a contradiction, as we shall see now. To evaluate the condition [*zx* > 0] (i.e. to fire the corresponding instruction), an event to which *x* belongs has to occur; but to produce *x* requires [*zx* > 0] to be evaluated; the result is a deadlock in the timing of the program. Note that a delay $ is present in the unique circuit of this graph, so that no starvation was expected because of short circuits in the data dependencies.

To summarize, the examples 4 and 6 we have investigated imply the following remarks:

REMARK 1: The examples above illustrate the importance of being able to reason about the *relative* timing of the various signals: this will be the purpose of the notion of clock we shall introduce hereafter.

REMARK 2: The latter two examples show that the *conditions* have to be handled with some care. In fact, since the *conditions* are the only instructions which transform the time in a way which depends upon the values of the signals, they must play a special role . We shall see later that the

situation of the example 4 above is the typical situation in which errors can be found in the timing of the program, whereas the example 6 exhibits a more subtle situation. As a conclusion, we need a formalism to mimic the reasoning which led to the discovery of a contradiction in the examples 4 and 6 above.

5.1.2 Clocks as equivalence classes of simultaneous signals.

We shall denote for short by

$$S(P) = ?P \cup !P \tag{5-1}$$

the set of all the interfaces of the process P. The elements of S(P) will be simply referred to as *signals*.

DEFINITION 3

(i) Given two elements a and b of S(P) , we shall say that a is less frequent than b, denoted by

$$a \subseteq b \tag{5-2}$$

if every transition of P involving $a[x]$ (with $x \neq \perp$) , involves also $b[y]$ (for $y \neq \perp$); $\subseteq$ is an order relation.

(ii) The associated equivalence relation is denoted by

$$a \equiv b \tag{5-3}$$

and their equivalence classes are referred to as *clocks*.□

COMMENT: The definition we have introduced for the notion of clock is explicitely intended to handle relationships between time histories, rather than the time histories themselves. This is the right way to take into account the inherent functional nature of the sequences of events (i.e. histories) in synchronous languages.

5.1.3 Clock transforms due to generators: an informal discussion.

Let us investigate informally how the generators of SIGNAL transform or constrain clocks. This investigation will be useful to introduce the formal model we shall use to analyse the timing of a SIGNAL process. Since the generators of SIGNAL have been precisely chosen to be matched with the elementary operations that can be expected on clocks, this informal derivation is easy, and will be left to the reader. Here are listed the generators, together with their effect on clocks:

- the *functions* : equality of the clocks of all ports
- the *$* : equality of the clocks of the input and output ports
- the *filter* : delivers the infimum of the input clocks
- the *merge* : delivers the supremum of the input clocks
- the *condition* tt : results in a new clock

The key remark is that the only relevant informations to reason about clocks is contained in the values *presence, absence*, or in the *values of boolean signals*. In the next paragraph, we shall introduce the convenient way to reason about the clocks of a SIGNAL process. This will be obtained through the introduction of a sublanguage of SIGNAL, called SIG, which contains the instructions that are useful to specify and analyse the synchronization of a SIGNAL process .

5.2 The sublanguage SIG.

The language SIG possesses two types: the type *boolean* and the type *dummy*. A boolean can take the values *tt* and *ff* as usually, whereas the type *dummy* possesses a single value, called *defined*, and denoted by *T*. Both types are as usually extended with the single value $\perp$, which means the absence of value. The language SIG possesses the following instructions

- the generic [2] function *synchro* which possesses only input ports;
- the usual boolean operations *and, or, not*, and their composition.
- **default, when, tt**
- relabelling, @, &.

Apart from the values of boolean expressions and the values of boolean memories, this mapping keeps all the information from a SIGNAL process, that is relevant to timing. To check the timing

---

2 with a parametrized list of t input values, and no output values

of a SIGNAL process P we shall map this process into the sublanguage SIG (see later). Before this, we shall show that SIG processes are very straightforward to analyse.

5.3 The mathematical model of SIG.

Recall the language SIG has two types of values: a three valued type ($\perp, tt, ff$) and a two valued type ($\perp$, T). Natural sets to represent these types in an effective way are respectively

- the set (0,1,2)
- the set (0,1)

To get an effective calculus on these sets, we shall endow the set (0,1,2) with the structure of the commutative field $Z/3Z$, and consider (0,1) as a subset of $Z/3Z$ [3]. Since the map

$$x \rightarrow x^2$$

maps respectively 0 onto 0 and (1,2) onto 1, we have the following

PROPERTIES :

(i) a function of dummy type is represented by a function of the form

$$x \rightarrow f(x^2) \tag{5-4}$$

for some function f from $Z/3Z$ into itself.

(ii) Since $x^3 = x$, any function on $Z/3Z$ is polynomial of degree 2 at most. □

Hence, we have the following illuminating proposition:

PROPOSITION 1: Any SIG process is represented by an algebraic manifold over the field[4] $Z/3Z$, where identical names refer to the same signals. □

This property is due to the fact that SIG does not possess the instruction $, so that a SIG process

---

[3] We shall sometimes write -1 instead of 2

[4] note that this field is not algebraically closed

has no memory, i.e. is purely static. The timing analysis of a SIG process is then reduced to the analysis of this manifold. For example, to check a timing error in a SIG process is equivalent to prove that a starvation occurs, i.e. that some coordinate, say x, of this manifold, is constrained to be always 0 (which means that the corresponding SIG value is always $\perp$). In the next paragraph, we shall see how to derive this manifold from the syntax of a SIGNAL program: this manifold will be called **the clock calculus** of the considered SIGNAL process.

5.4 The clock calculus of a SIGNAL process.

5.4.1 The clock calculi of the generators of SIGNAL.

These calculi will be presented according to the following syntax: given a generator, say with input ports a, b and output ports x, y, z, its clock calculus will be a system of equations of the generic form

$$f(a, b, x, y, z) = 0$$

where $a, x, \ldots$ denote respectively the images in $Z/3Z$ of the signals with the corresponding names, and f is a function on $Z/3Z$. Here are listed the clock calculi of the generators of SIGNAL; in this table, "non bool function" refer to any function which is not obtained by composing the boolean functions *and, or, not.*

Table 1. CLOCK CALCULI OF THE GENERATORS

Generator	Clock calculus
$y := \text{not}\, x$	$y = -x$
$y := a \text{ or } b$	$y = ab(1 - (ab + a + b))$
$y := a \text{ and } b$	$y = ab(ab - (a + b + 1))$
$x := \$y$	$x^2 = y^2$
non bool function	$x^2 = y^2 \quad \forall x, y \in S(P)$
$x := a \text{ when } b$	$x = ab^2$ if x boolean $x^2 = a^2b^2$ otherwise
$x := a \text{ default } b$	$x = a + b - a^2b$ if x boolean $x^2 = a^2 + b^2 - a^2b^2$ otherwise
$h := \text{tt}(C)$	$h = -C - C^2$

JUSTIFICATION: Recall that S(P) is the set of the signals of the process P (see (5-1). The principle of the justification is as follows: map all the SIGNAL generators onto SIG generators. Then the proof rests upon a simple although tedious verification of the formulas of the table 1. We shall illustrate this through the proof of some of these formulas. First, the map $x \to -x$ maps the triple $(0,1,-1)$ onto the triple $(0,-1,1)$, which represents the boolean function *not*; the calculi of the other boolean functions can be verified in the same way. Non boolean functions result only in synchronisation of all ports, i.e. such functions are mapped into functions **synchro** possessing as input ports all the ports of this function. This is exactly what the equation of the "non bool function" expresses, since this equation only requires values to be present simultaneously. The same argument holds for the $. Finally, the other formulas are easy to justify; note that the non boolean formulas are just obtained by taking the square of both handsides of the corresponding boolean formulas, which is in accordance with (5-4). □

REMARK: One should remember that the same name can be used to denote an input or an output port, like in the expression

$$x := a + x$$

in which case $?x$ and $!x$ must be distinguished in the corresponding clock calculus.

5.4.2 Clock transfers due to the interconnection operators.

The formulas we shall give can be used to build step by step the clock calculus of a SIGNAL process. To present these formulas, we shall need the following notations. Given a process **P**, we shall denote by

$$EQ\{\mathbf{P}\}$$

the clock calculus of the process **P**. Given two calculi (i.e. two sets of equations) $EQ1$ and $EQ2$, we shall denote by

$$EQ1 \cup EQ2$$

the union of these sets, where identical names (including the prelabelling "?" or "") refer to the same signals. To refer to the calculus obtained by substituting the name *b* for the name *a* in a calculus EQ, we shall write .

$$EQ\ a{:}b$$

The clock transfers due to interconnection operators are listed below.

Table 2. CLOCK TRANSFERS

connection operator	clock transfer
P ?a:b	$EQ\{\mathbf{P}\}$?a:?b
P !x:y	$EQ\{\mathbf{P}\}$!x:!y
P & **Q**	$EQ\{\mathbf{P}\} \cup EQ\{\mathbf{Q}\}$
P @ x	$EQ\{\mathbf{P}\}$ if x $\notin$!**P** $EQ\{\mathbf{P}\}$?x:!x otherwise

These transfer formulas are intuitively justified. Recall that the use of the clock calculus will be mathematically sounded in the chapter 7.

CONCLUSION: We have introduced the clock calculus of a process as a tool to analyse the timing. This was obtained in the following way:

1. the sublanguage SIG has been introduced, which is the purely static sublanguage of SIGNAL that keeps the largest part of the synchronization mechanisms, and an effective map from SIGNAL onto SIG has been introduced to summarize the synchronization mechanism of any SIGNAL process;
2. it has been shown that any SIG process is completely represented by a system of static algebraic equations over the commutative field $Z/3Z$, called its *calculus.*

These tools will be illustrated on the examples above. But we shall before introduce the reader to the algebraic calculus on the field $Z/3Z$.

5.5 Solving clock calculi.

5.5.1 The equation $aX^2 + bX + c = 0$.

The solution of this equation will be different according to the expected type of X, namely *boolean* or *dummy*, since in the latter case we have seen that the solution must be of the form $X^2 = ...$

WARNING: Since primitive boolean expressions always carry free values, only presence/absence can be constrained for such signals. Hence, when X is a primitive boolean expression, the equation $aX^2 + bX + c = 0$ must be solved for X^2.

PROPOSITION 2

(i) If X is *boolean*, the equation $aX^2 + bX + c = 0$ is equivalent to the two equations

$$c[(a+c)^2 - b^2] = 0 \quad (i)$$
$$X = \Phi(1-a^2)(1-b^2)(1-c^2) - bc(1-a^2) + a(b \pm \delta) \quad (ii) \qquad (5\text{-}5)$$

where δ is the discriminant, given by

$$\delta = (b^2 - ac)^{1/2}$$

and Φ is a free parameter, called *phantom.* Not that the equation (5-5-i) can also be rewritten

$$a^2c - ac^2 - b^2c + c = 0 \tag{5-6}$$

(ii) If X is *dummy* or is a primitive boolean expression, the equation $aX^2 + bX + c = 0$ is equivalent to the two equations

$$\begin{aligned} acb^2 - a^2b^2 + ac + c^2 &= 0 \quad (i) \\ X^2 = \Phi^2(1 - a^2)(1 - b^2)(1 - c^2) - ac &\quad (ii) \end{aligned} \tag{5-7}$$

where Φ is a phantom.□

PROOF: Let us begin with the proof of (ii), which is easier. First, recall that, when X is *dummy*, or is a primitive boolean expression, we must solve the equation directly in terms of X^2. As a consequence, the equation $aX^2 + bX + c = 0$ must be of one of the following forms, where the written coefficients are nonzero:

$$\begin{aligned} bX &= 0 \quad (i) \\ aX^2 &= 0 \quad (ii) \\ aX^2 + c &= 0 \quad (iii) \end{aligned} \tag{5-8}$$

For (5-8-iii) to hold, the discriminant, which is now equal to $-ac$, must be a square, which is equivalent to require that $-ac \neq -1$, or, equivalently, when $a \neq 0$,

$$ac(c + a) = 0 \tag{5-9}$$

Finally, at least one of the following constraints must be satisfied

$$\begin{aligned} a = c &= 0 \quad (i) \\ b = c &= 0 \quad (ii) \\ b = 0 \text{ and } ac(c + a) &= 0 \quad (iii) \end{aligned} \tag{5-10}$$

To prove that this is equivalent to (5-7,i), we use the following lemma, the proof of which is immediate:

LEMMA 1: the following formulas hold:

$$\{p = 0 \text{ and } q = 0\} \Leftrightarrow \{p^2 + q^2 = 0\} \tag{5-11}$$

$$\{p = 0 \Rightarrow q = 0\} \Leftrightarrow \{q(p^2 - 1) = 0\} \ \square \tag{5-12}$$

Applying the lemma to the constraints 5-10 yields

$$(a^2 + c^2)(b^2 + c^2)(b^2 + a^2c^2(c + a)^2) = 0 \tag{5-13}$$

which is equivalent to (5-7-i). The proof of (5-7-ii) is then straightforward. This proves the formula for the case (ii) . The proof of (i) goes along a similar way, although with more tedious calculations; hence we shall omit the details. First,

$$\{a = b = 0\} \Rightarrow \{c = 0\} \tag{5-14}$$

Then, the discriminant must be a square:

$$\delta^2 = b^2 - ac \neq -1 \tag{5-15}$$

Thanks to the lemma 1, combining 5-14 and 5-15 gives (5-5-i). On the other hand, X equals the first term on the right handside of (5-5-ii) when $a = b = 0$, the second one when $a = 0$ but $b \neq 0$, and the third one when $a \neq 0$. This finishes the proof of the proposition. □

5.5.2 Solving systems of equations.

As the analysis of the examples will show, the synchrony constraints caused by the functions are generally at the origin of *implicit* relationships between the clocks of a SIGNAL process. These are translated into *implicit* systems of equations in the clock calculus of the considered process. Hence the following definition:

DEFINITION 4: Solving a clock calculus is, by definition, calculating a minimal parametrization of the corresponding algebraic manifold.

Recall that, given an algebraic manifold defined by a system of polynomial equations in the variables $X_1, \ldots, X_n$, say

$$
\begin{aligned}
P_1(X_1,\ldots,X_n,a_1,\ldots,a_m) &= 0 \\
&\ldots \\
&\ldots \\
P_k(X_1,\ldots,X_n,a_1,\ldots,a_m) &= 0
\end{aligned}
\tag{5-16}
$$

where $a_1,\ldots,a_m$ are parameters or constants, a *parametrization* of this manifold is another system of polynomial equations equivalent to (5-16), which is of the form

$$
\begin{aligned}
X_1 &= Q_1(Y_1,\ldots,Y_p,a_1,\ldots,a_m) \\
&\ldots \\
&\ldots \\
X_n &= Q_n(Y_1,\ldots,Y_p,a_1,\ldots,a_m)
\end{aligned}
\tag{5-17}
$$

where the Q_i's are polynomials, and the Y_i's are the free parameters. Obviously, some of the X_i's can be chosen as these free parameters. This parametrization is called *minimal* if the set of the so introduced free parameters has minimal cardinality; minimal parametrizations are not unique, but still represent the same manifold. We shall now sketch a possible algorithm to solve a given clock calculus.

STEP 0: Start with the set of equations (5-16), we shall denote for short by $EQ(X_1,\ldots,X_n)$.

STEP 1: Partition $EQ(X_1,\ldots,X_n)$ as

$$EQ(X_1,\ldots,X_n) = EQ(X_2,\ldots,X_n) \cup EQ'$$

where $EQ(X_2,\ldots,X_n)$ refer to the subset of the equations of $EQ(X_1,\ldots,X_n)$ that do not involve X_1.

STEP 2: Replace EQ' by a single equation $EQ[X_1]$ using (5-11) repeatedly.

STEP 3: By proposition 2,

$$EQ[X_1] \Leftrightarrow \{D[X_1], C(X_2,\ldots,X_n)\}$$

where $D[X_1]$ is a *definition* of X_1 according to (5-5-ii) or (5-7-ii), and $C(X_2,\ldots,X_n)$ is the

associated *constraint* according to (5-5-i) or (5-7-i) respectively; this constraint does not involve the variable X_1. Note that the implicit form $EQ[X_1]$ can be kept as well as a definition of X_1.

STEP 4: Go back to step 0 with

$$EQ(X_2,...,X_n) \cup C(X_2,...,X_n)$$

instead of $EQ(X_1,...,X_n)$. □

Finally, free parameters of the solved calculus will be

- the values $\{+1,-1\}$ of all primitive boolean expressions,
- a subset of the input signals,
- phantom signals, such as the Φ in the formulas 5-5 and 5-7.

We have sketched a possible effective algorithm to solve the clock calculus of a process. We do not claim that this is the most efficient one, but this algorithm clearly always terminates, and requires no backtracking. We shall now illustrate this procedure on the examples of the next section .

5.6 Back to the examples.

5.6.1 Example 2, continued.

Using the tables 1 and 2 and keeping in mind that a is non boolean, we get the primitive clock calculus

$$x^2 = a^2 + b^2 - (ab)^2$$
$$z^2 = x^2 = a^2$$

Elementary substitutions yield

$$(ab)^2 - b^2 = 0$$
$$z^2 = a^2$$
$$x^2 = a^2$$

The calculus exhibits one cycle, corresponding to the first equation. Solving for b and applying the proposition 2 yields

$$0 = 0 \quad \textit{for the constraint}$$
$$b^2 = -\Phi^2((a^2 - 1)^2 + 2) = \Phi^2 a^2$$

where Φ is a phantom; but this equation means that b is less frequent than a, which was the expected result.

5.6.2 Example 4, continued.

Here, the signals a and b are non boolean. The primitive clock calculus is

$$z^2 = x^2 = b^2 \quad (i)$$
$$[a < b]^2 = a^2 = b^2 \quad (ii)$$
$$h = -[a < b] - [a < b]^2 \quad (iii)$$
$$x^2 = a^2 h^2 \quad (iv)$$

The equation (i) is due to the function + ; the equation (ii) comes from the primitive boolean expression $[a < b]$; the equation (iii) is the result of the instruction tt ; finally, the equation (iv) is the equation of the when for non boolean signals. A substitution yields the implicit equation

$$[a < b]^2 = [a < b]^2(-[a < b] - [a < b]^2)$$

which is of the form

$$X^2 - X = 0$$

Since X is here a primitive boolean expression, its defined values must be free, which means that this equation must be solved for X^2, using the formula 5-7; this yields

$$1 = 0$$

for the constraint (5-7-i), which is obviously violated. The program is then rejected.

5.6.3 Example 6, continued.

The input a is non boolean; the primitive clock calculus is

$$h = -[zx > 0] - [zx > 0]^2 \quad (i)$$
$$[zx > 0]^2 = zx^2 \quad (ii)$$
$$x^2 = zx^2 \quad (iii)$$
$$x^2 = a^2h^2 \quad (iv)$$

Again, a substitution yields the implicit equation

$$[zx > 0]^2 = a^2(-[zx > 0] - [zx > 0]^2)$$

which is of the form

$$X^2 = a^2(-X - X^2)$$

This equation can then be solved for X, or for a. If we chose a as a parameter, we must solve for X^2 since X is a primitive boolean expression the defined values of which cannot be constrained. The constraint (5-7-i) is here

$$(a^2 + 1)a^2 = 0$$

which implies

$$a = 0$$

a constraint which means a total starvation of the process: this program is then interpreted as incorrect. On the other hand, solving for the non boolean a yields the constraint

$$((X^2 + X)X^2)^2 + (X^2 + X)X^2 = 0$$

which again implies

$$X = 0$$

i.e. a total starvation. The program is rejected. These examples illustrate the power of the clock calculus in analysing the timing of a SIGNAL program.

5.6.4 Example 7.

This new example illustrates how the clock calculus can prove the equivalence of programs with significantly different syntaxes, which should be obviously accepted as equivalent by the programmer. Let *a* and *b* be two non boolean input signals; we would like to prove the equivalence of instructions of the extended language SIGNAL like

if a<0 and b<0 then ...

and

if a<0 then if b<0 then ...

These instructions are respectively expanded in the primitive language SIGNAL as follows

((C := [a<0] and [b<0]) & (h := tt(C))) @ C

((C1 := [a<0]) & (h1 := tt(C1)) & (C2 := [b<0] when h1) & (h := tt(C2))) @ C1, h1, C2

We shall write for short a and b instead of [a<0]

The clock calculus of the first program is

$$C = ab(ab - a - b - 1)$$
$$h = -C - C^2$$

which yields

$$h = a^2b^2 + a^2b + ab^2 + ab \qquad (5\text{–}18)$$

On the other hand, the clock calculus of the second program is

$$C1 = a$$
$$h1 = -C1 - C1^2$$
$$C2 = b.h1^2$$
$$h = -C2 - C2^2$$

which yields also 5–18. This proves the equivalence of the two programs, since both specify only synchronisation.

Chapter Six
DATA DEPENDENCIES.

The purpose of this section is to analyse the instantaneous data dependencies. Our goals are

- to detect short circuits, i.e. non computable transitions: if non computable transitions are present, the corresponding process is said to be *non computable*, whereas it is said to be *computable* otherwise;
- if the considered process is computable, to produce its dependence graph at any of its transitions.

6.1 Further examples

6.1.1 Example 8.

Consider the following program, where s is a parameter:

P = ((h := tt[y<s]) & (y' := y **when** h) & (x := 0.5 z + y') & (z := $x)) @ h,y',z,x

This program selects the instants at which y<s and generates at the corresponding clock the recurrent equation x := 0.5 x + y with the values of y available at these instants. In the classical methods of dependency analysis for single clocked recurrent equations (see for example [Oppenheim & Schafer, 1975]), memories appear as source nodes of the dependence graph. But, in the present case, the memory is read only when the condition y<s is true, so that this condition has to be evaluated before to fire the memory of the $. This clearly makes impossible for this memory to be a source node of the classical dependence graph.

6.1.2 Example 9.

Consider the following program:

P = ((y := x **when** h) & (u := y + z) & (z := $u) & (x := a **default** u)) @ x,y,u,z

This is a much more subtle example. The reader can check that there are several possible correct timings for this program (solve the clock calculus!). The most frequent timing for x is the supremum of the clocks of a and h, whereas the less frequent is the clock of a. If the clock of a

is chosen, this means that the value of u is never chosen to compute x, so that *there is no short circuit in this case.* But, if the most frequent one is chosen, then u will sometimes be used to compute x, which causes a short circuit! This example clearly shows that the notion of short circuit cannot be independent from the timing; in the present example, a short circuit would be always detected with naive methods, this problem was indeed pointed out in [Berry and Cosserat, 1984].

6.2 The conditional dependence graph.

6.2.1 Definition and basic properties.

Let **P** be a process specified according to the notations of (2-5) . We shall associate to **P** its *complete conditional dependence graph*, denoted by $CDG\{\mathbf{P}\}$ as follows.

DEFINITION 5: $CDG\{\mathbf{P}\}$ is a labelled oriented graph defined as follows.

(i) The nodes of $CDG\{\mathbf{P}\}$ are

- the elements of $S(\mathbf{P})$
- the values *omem* and *nmem* carried by the memories of **P** before and after its transitions.

(ii) Given two nodes a and b of $CDG\{\mathbf{P}\}$ we shall write

$$a \xrightarrow{h} b \tag{6-1}$$

if h is the largest clock $\subseteq a$ [5] such that a *influences* b, i.e., using the notations of (2-4), and the notations of the clock calculus,

$$\{h^2 \neq 0\} \quad \Rightarrow \quad \{a[x] \quad \Rightarrow \quad b[y(x)]\} \quad \square \tag{6-2}$$

Obviously, if $h \equiv \phi$ we do not write any branch originating from a and terminating at b. We shall

5 see (5-2) for the definition of this order relationship on clocks

now give a fundamental transitivity property of this labelled graph, that will allow us to calculate all the branches from a minimal subset of them. Before to state this result, we shall introduce the following notations, where the polynomial expressions refer to the clock algebra introduced in the preceding section.

$$\{x \equiv a \otimes b\} \Leftrightarrow \{x^2 = a^2 b^2\} \quad (i)$$

$$\{x \equiv a \oplus b\} \Leftrightarrow \{x^2 = a^2 + b^2 - a^2 b^2\} \quad (ii)$$

$$\{\overset{o}{\Pi}\} \Leftrightarrow \{\textit{iteration of} \otimes\} \quad (iii)$$

$$\{\overset{o}{\Sigma}\} \Leftrightarrow \{\textit{iteration of} \oplus\} \quad (iv) \qquad (6\text{-}3)$$

Note that $\otimes$ is nothing but the infimum of clocks according to the order (5-2), whereas $\oplus$ is the corresponding supremum, see the definition 3, so that these operations satisfy the usual properties of the lattice operations. Using these notations, we have the following proposition, the proof of which is obvious:

PROPOSITION 3: The following transitivity property holds: suppose there is a path

$$a \xrightarrow{h} z \qquad (6\text{-}4)$$

Then, if we consider all the branches originating at a and terminating at z, say:

$$DP_i: \; a \xrightarrow{h_{i;1}} n_{i;1} \; \dots \; n_{i;j-1} \xrightarrow{h_{i;j}} n_{i;j} \; \dots \rightarrow z \qquad (6\text{-}5)$$

then the following corresponding decomposition of the clock h holds:

$$h \equiv \sum_{i}^{o} \prod_{j}^{o} h_{i;j} \quad \square \tag{6-6}$$

PROOF: (6-5) implies that $h_i \subseteq h$, where h_i is defined by

$$h_i \equiv \prod_{j}^{o} h_{i;j}$$

since, by transitivity, a influences z under the clock h_i. Consequently,

$$\sum_{i}^{o} \prod_{j}^{o} h_{i;j} \subseteq h.$$

But $\equiv$ holds in fact, since the path $a \rightarrow z$ is itself one of the pathes DP_i. $\square$

Note that, since the operation $\oplus$ is idempotent, the fact that the original branch does itself belong to the CDG_i's has no importance. As we shall see, this proposition will allow us to simplify the construction of the complete conditional dependence graph.

DEFINITION 6: Given a process **P** and its complete conditional dependence graph $CDG\{\mathbf{P}\}$, we shall say that a labelled graph DG *generates* $CDG\{\mathbf{P}\}$ if

1. the nodes of both graphs are the same;
2. for every branch (6-4) of $CDG\{\mathbf{P}\}$ the decomposition (6-6) holds, where the pathes (6-5) are all the pathes *in the graph* DG that originate from a and terminate at z.

The proposition 3 allows us to replace the construction of the graph $CDG\{\mathbf{P}\}$ by the construction of any generating graph. Any graph which generates $CDG\{\mathbf{P}\}$ will be called a *conditional dependence graph* of the process **P**, and will be denoted generically by $DG\{\mathbf{P}\}$. Another basic property is the following, which expresses that the notion of conditional dependence graph is the key tool to detect short circuits in a SIGNAL process. In the following theorem, the word *circuit* denotes as usually a non trivial path originating at a node a and terminating at the same node.

THEOREM 4: A process **P** is computable if and only if there exists a $DG\{\mathbf{P}\}$ satisfying the so-called *circuit-free* property we state now: for every circuit

$$a \xrightarrow{h_1} \ldots \xrightarrow{h_n} a \tag{6-7}$$

the following equality holds:

$$\prod_{i=1\ldots n}^{o} h_i \equiv \phi \ \square \tag{6-8}$$

PROOF: First, note that, if some graph $DG\{\mathbf{P}\}$ is circuit-free, then any other generating graph is also circuit-free. Now, assume that there exists a generating graph which is not circuit-free, and let a be a node of this circuit. Denoting by x the value carried by a, using the proposition 3 and the definition 5, we get that

$$a[x(x)] \tag{6-9}$$

i.e. the value x is a static function of itself in some transition of **P**, a situation which is forbidden, as we have stated in the presentation of the static language; this is exactly what we call a "short-circuit". Conversely, if every circuit of the generating graph violates the condition (6-8), then it is not possible for the condition (6-9) to hold with $x \neq \perp$, which means that there is no short-circuit.

CONSEQUENCE: *If a process* **P** *possesses a circuit-free conditional dependence graph, then all the p-connections that were used to construct it are valid* (see the discussion of the rule of the p-connection), and **P** is computable.

Let us emphasize that, although the theorem 4 gives a necessary and sufficient condition, it might happens that (6-8) actually holds, while our clock calculus be unable to prove it. The next paragraphs will be devoted to the presentation of effective algorithms to construct a graph $DG\{\mathbf{P}\}$.

6.3 Derivation of the conditional dependence graph.

6.3.1 The conditional dependence graph of the generators.

Here are listed the conditional dependence graphs of the generators.

Table 3. CONDITIONAL DEPENDENCE GRAPH OF THE GENERATORS

Generator	Conditional dependence graph
function	dependence graph of the corresponding static function labelled with the clock of the function
$y := \$x$	$o\$x \xrightarrow{?x} !y$ $?x \xrightarrow{?x} n\$x$
$x := a$ **when** h	$?a \xrightarrow{a \otimes h} !x$
$x := a$ **default** b	$?a \xrightarrow{?a} !x$ $?b \xrightarrow{?b \ominus ?a} !x$
$h := \mathrm{tt}(C)$	no contribution

COMMENTS: The notations $o\$x$ and $n\$x$ denote the values carried by the memory of the instruction $y := \$x$ respectively before and after the transition. The clock of a function is obviously the clock of any of its signals. The operation $\ominus$ is defined in terms of the clock algebra by

$$\{x \equiv a \ominus b\} \Leftrightarrow \{x^2 = a^2 - a^2 b^2\} \tag{6-10}$$

Finally, the instruction **tt** does not contribute to the dependence graph, since no input value influences the output in the instruction *condition*. The proof of these formulas are straightforward.

6.3.2 Transfers due to interconnection operators.

The purpose of this paragraph is to derive formulas that will allow to construct a generating conditional dependence graph of any process, in a way similar to the one followed to derive the clock calculus of a process. In the sequel, we shall use the following notations. Given a labelled graph DG, we shall denote by

$$DG\ a{:}b \tag{6-11}$$

the labelled graph obtained by substituting *at the nodes and at the labels* the name *b* for the name *a*. The notation

$$DG1\ \cup\ DG2 \tag{6-12}$$

denotes the union of the two graphs $DG1$ and $DG2$ where common names refer to the same objects. Using these notations, we can state the following formulas.

Table 4. DEPENDENCE GRAPH TRANSFERS

connection operator	graph transfer
P ?a:b	$DG\{$**P**$\}$?a:?b
P !x:y	$DG\{$**P**$\}$!x:!y
P & **Q**	$DG\{$**P**$\}$ ∪ $DG\{$**Q**$\}$
P @ x	$DG\{$**P**$\}$ if x ∉ !**P** $DG\{$**P**$\}$?x:!x otherwise

PROOF: The proof goes along the following lines. We assume that $DG\{$**P**$\}$ is a generating conditional dependence graph for **P**; then, we prove that the modified graph indicated in the right column is also a generating dependence graph of the new process resulting from the interconnection operator. Consider first the transfers due to the relabellings. The proof is trivial for the output relabelling, since its effect is purely a change of name; the same holds for the input relabelling when b was not an input port of the process **P**. Consider then the case of the input relabelling when b is an input port of **P**; the resulting data dependencies is obtained by

identifying the nodes ?a and ?b in $DG\{\mathbf{P}\}$, which is the formula of the table; on the other hand,the transfers on the labels are exactly those obtained through the clock transfer formulas of the table 2 which are identical to the present ones for the labels of the branches; this proves the formulas for the relabellings. The proof of the formula of the collateral is exactly the same as for the clock transfers formulas of the table 2. The proof of the formulas of the p-connection is less trivial. To show that $DG\{\mathbf{P}\}$?x:!x generates $CDG\{\mathbf{P}$ @x$\}$, we must show that the condition (2) of the definition 6 is satisfied. Assume we have in $CDG\{\mathbf{P}\}$ the branches

$$a \xrightarrow{k1} !x \qquad (i)$$
$$?x \xrightarrow{k2} z \qquad (ii)$$
$$a \xrightarrow{k} z \qquad (iii) \qquad (6\text{-}13)$$

The proposition 3 implies that we have in $CDG\{\mathbf{P}$ @x$\}$

$$a \xrightarrow{h} z$$
$$h \equiv (k1 \otimes k2) \oplus k \qquad (6\text{-}14)$$

Finally, the graph transfer formula of the p-connection, the decomposition (6-14), and the fact that, by induction, the condition (2) of the definition 6 was satisfied for the branches listed in (6-13), imply together that the branch considered in (6-14) also satisfies the condition (2) of the definition 6. □

CONCLUSION: we have derived an effective algorithm to construct a generating conditional dependence graph for any process. In the sequel, *dependence graph* will be used for short instead of *generating conditional dependence graph.*

6.4 Back to the examples.

In the forthcoming examples, we shall sometimes partially solve the clock calculus to let the labels of the branches to be easier to interpret.

6.4.1 Example 8, continued

The preceding formulas yield the dependence graph shown in the figure 3 below.

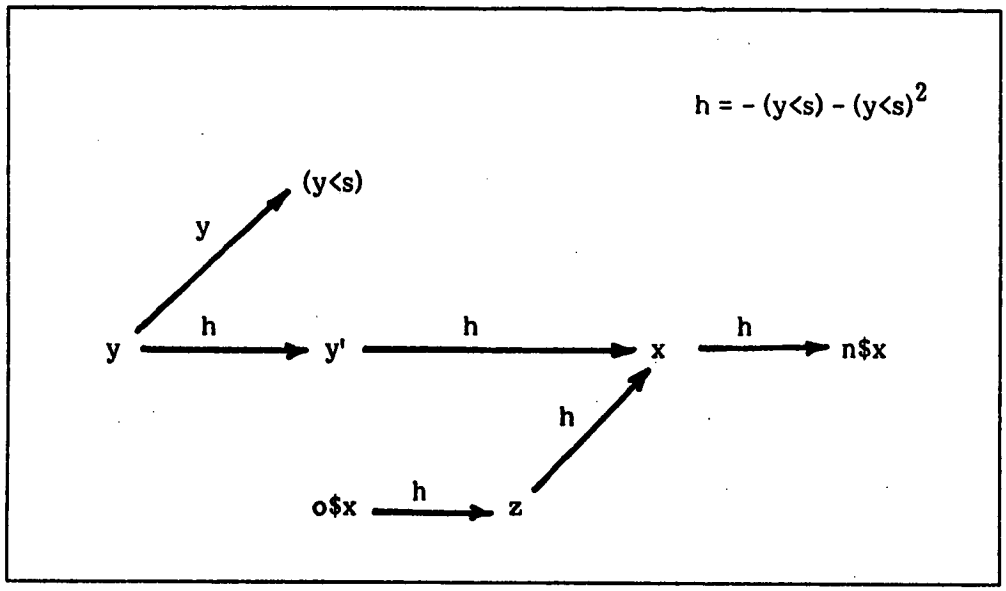

Figure 3. Dependence graph of the example 8

This dependence graph is circuit-free; note that the branch originating from the memory o$x is labelled by the clock h , which is known only once the condition [y<s] has been evaluated.

6.4.2 Example 9, continued

The dependence graph of this process is shown in the figure 4 below.

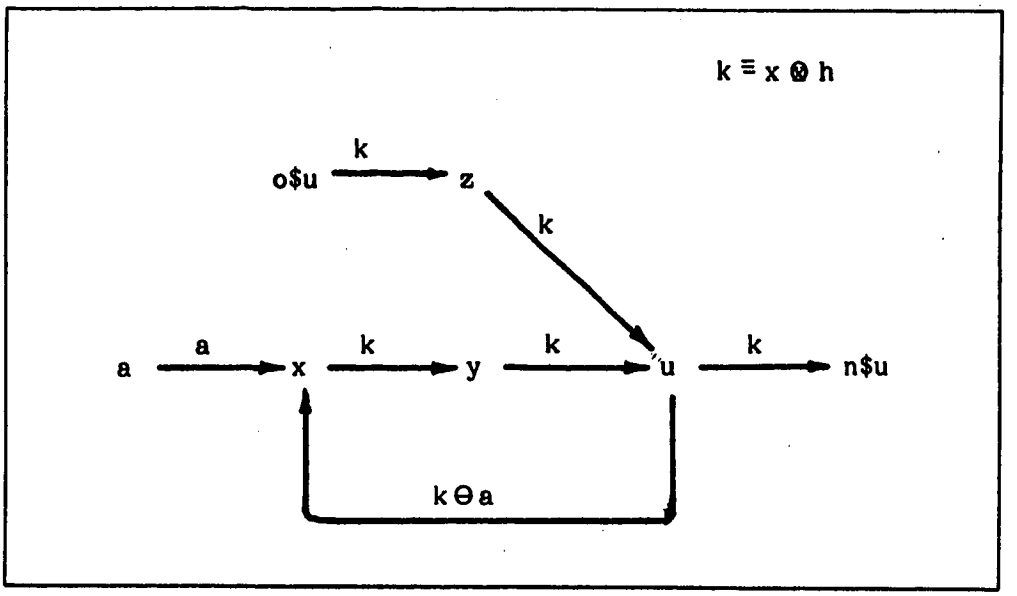

Figure 4. Dependence graph of the example 9

In this figure, we have not solved the clock calculus. This graph exhibits a cycle, so that we must check the condition of the theorem 4 on this cycle to know whether or not this process is really circuit-free. To do this, we must solve the clock calculus; we shall only indicate the main steps of this resolution. Simple substitutions yield the following formulas, where we don't indicate the prelabels "?,", since no confusion is possible.

$$u^2 = x^2 h^2$$
$$x^2 = u^2 + a^2 - a^2 u^2 \tag{6-15}$$

Solving this calculus for x yields

$$x^2 = a^2 + \Phi^2 (h^2 - a^2 h^2) \tag{6-16}$$

where Φ is a phantom. Using (6-3), the condition (6-8) is written

$$h^2 x^2 u^2 (u^2 - a^2 u^2) = 0 \tag{6-17}$$

Using (6-15) and (6-16), the left handside of (6-17) is equal to

$$\Phi^2(h^2 - a^2h^2) \tag{6-18}$$

But (6-18) means that *the phantom Φ must be chosen equal to 0 in order for the conditional dependence graph to be circuit-free* . This is exactly the condition we expected.

6.4.3 Example 10

Consider the following program we should write in an extended version of SIGNAL as

(if a<b then y := (a+b) default x else x := (a-b) default y fi) @ x,y

This program is rewritten in the primitive language SIGNAL as

```
( (h := tt([a<b]) ) & (k := tt(not [a<b]) )

& (apb := a+b) & (z := apb when h) & (y := z default x)

& (amb := a-b) & (u := amb when k) & (x := u default y) )

@ h,k,apb,z,amb,u,x,y
```

The corresponding dependence graph is shown in the figure 5 below, where "c" denotes for short the boolean expression "[a<b]".

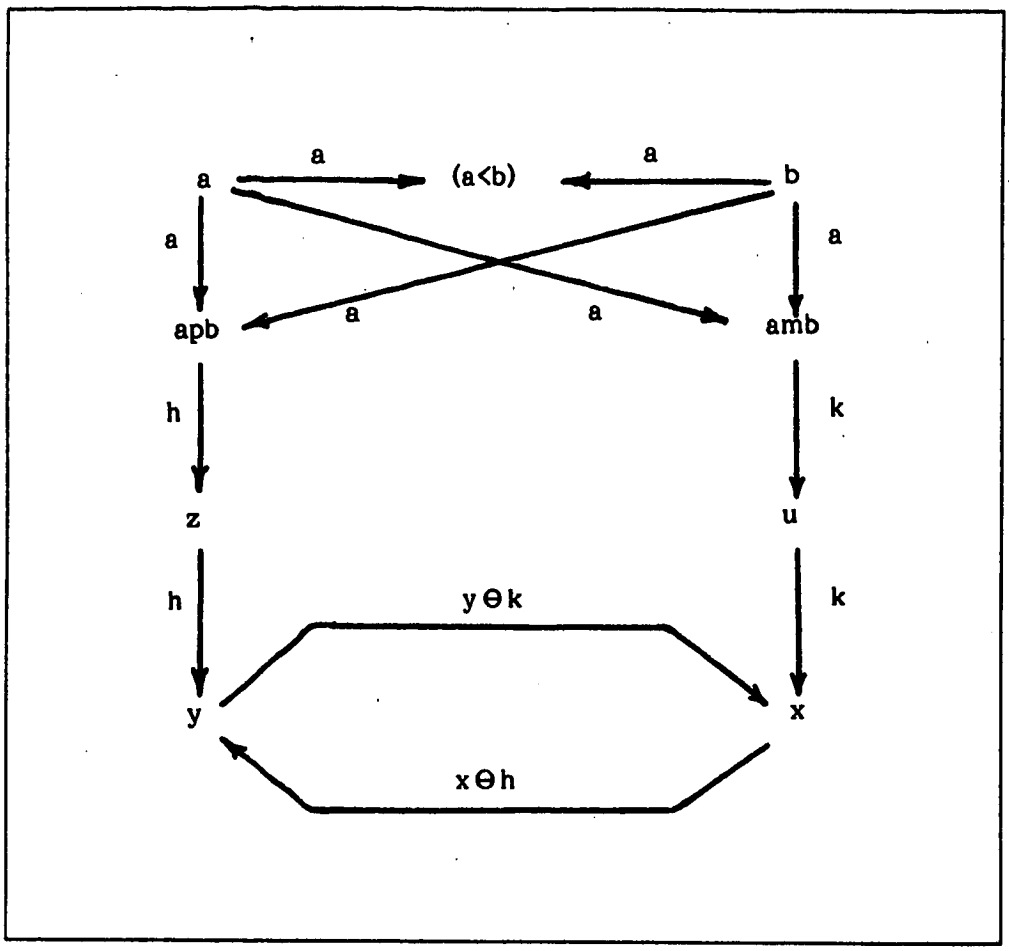

Figure 5. Dependence graph of the example 10

In this figure, the clock calculus has not been solved. This graph apparently exhibits a cycle, so that we must check the condition of the theorem 3. To do this, we must before solve the clock calculus. The primitive clock calculus is

$$a^2 = b^2 = amb^2 = apb^2 = c^2$$
$$h = -c - c^2$$
$$k = c - c^2$$
$$z^2 = a^2h^2$$
$$u^2 = a^2k^2$$
$$y^2 = z^2 + x^2 - z^2x^2$$
$$x^2 = u^2 + y^2 - u^2y^2$$

Solving this calculus for x and y yields

$$x^2 = y^2 = a^2 + \Phi^2(1 - a^2) \tag{6-19}$$

where Φ is a phantom, so that the cyle of the figure gives for the condition (6-8) the resulting clock

$$\Phi^2(1 - a^2) \tag{6-20}$$

Again, the dependence graph is circuit-free if and only if the phantom Φ is chosen equal to 0. The reader can check that this is indeed what the programmer should expect, since this program can be executed if no parasitic communication requests between x and y are introduced. Note that this kind of program is rejected as incorrect by the causality checking algorithm of the language ESTEREL (see [Berry & Cosserat, 1984]); the reason being that the dependence graph used in this language is not a conditional one.

Chapter Seven
MAIN THEOREMS.

The purpose of this chapter is to prove mathematically the claims of the chapter on the clock calculus. By the way, we shall introduce a poweful method which will provide us with the convenient tools to justify a large class of clock calculi, among which the clock calculus we have chosen to use is only a possible one. This key tool will be the notion of *cut* we shall introduce and analyse now.

7.1 Cuts, and related basic lemmas.

7.1.1 Cuts.

DEFINITION 6 : Let us consider a process P { $P ?P !Q,$p_1,\ldots,p_k$}, where !Q denotes an unspecified list of output ports, the remaining output ports being the p_i's. Then, the process

$$Q\ \{\ \$P\ ?P, *p_1,\ldots,*p_k\ !Q\ \}\ =\ P\mathrm{cut}\,p_1,\ldots,p_k \tag{7-1}$$

is defined by the rule

$$\frac{P:\ \ <\$> \xrightarrow[\ !Q\ p_1[x_1:=\exp_1]\ \ p_2[\perp]\ \ldots]{?P} <\$'>}{Q:\ \ <\$> \xrightarrow[\ !Q]{?P\ *p_1[x_1]\ *p_2[\perp]\ \ldots} <\$'>} \tag{7-2}$$

where the $*$'s are simply markers to distinguish the corresponding ports from possibly preexisting input ports with the same names p_i's. In this rule, the usual shortages have been used. In particular, the reader should understand that the $\perp$'s are located at the same indices $i = 1,\ldots,k$ at the numerator and at the dénominator. □

Roughly speaking, the dependence of the output values x_i in P has been broken to create in Q corresponding free values, hence the name of *cut*.

These transfer formulas will be used in the sequel to build complex cuts from elementary cuts defined on generators.

7.1.3 Basic properties of cuts.

7.1.3.1 Computable transitions.

Let P and Q be as in the definition 6, and denote by cut(T) the transition of Q associated to the transition T of P by the rule (7-2). Then, the following result holds, the proof of which is immediate:

LEMMA 2: If the transition T is computable, so is the transition cut(T). □

7.1.3.2 Time-correctness.

Let P and Q be as in the definition 6. Here follows the basic result about cuts.

LEMMA 3: Assume every transition of P to be computable. Let $\alpha = \{a_1, ..., a_n\}$ be a time-incorrect subset of input ports of P. Then, α is the intersection with ?P of a time-incorrect set of input ports in Q. □

PROOF: Assume the conclusion does not hold, i.e.

$$\forall \pi \subset \{p_1, ..., p_k\}, \forall \xi \text{ candidate for } \alpha U \pi, \forall o\mu,$$
$$\exists \Xi \text{ candidate}, \xi \subset \Xi, \text{ such that an acceptance}$$
$$\text{using a non trivial subset of } \xi \text{ can be extracted from the pair } (o\mu, \Xi)$$

But (7-2) implies that the corresponding transition for Q must be derived from some transition for P which acts on the ports belonging to ?P in the same way for P and Q; in particular this transition must use in some non trivial fashion the values proposed to α. Since this holds for arbitrary candidate values, α cannot be time-incorrect in P, a contradiction. This proves the lemma.

LEMMA 4: If Q is fully time-correct, and if every transition of P is computable, then P is fully time-correct. □

PROOF: A direct consequence of the lemma 3.

7.1.3.3 Determinism.

Again, let P and Q be according to the definition 6. The following lemma is obvious.

LEMMA 5: If the set of the allowed transitions of Q is always a singleton, then the same property holds for P. □

Finally, the following lemma is a direct consequence of the previous results.

LEMMA 6: Assume that every transition of P is computable.

(i) Then, if Q is fully time-correct, and if the set of the allowed transitions of Q is always a singleton, then P is fully time-correct and deterministic.

(ii) If Q is fully time-correct, and if the set of the i-o-allowed transitions of Q is always a singleton, then P is fully time-correct and weakly deterministic. □

Note that Q being deterministic does not imply that the set of its allowed transitions be always a singleton, since we don't know a priori that all the allowed transitions are effectively eventually used (there are only sufficient conditions for the determinism, but no necessary and sufficient conditions).

7.2 Proving the clock calculus.

7.2.1 The map CLOCK defined on SIGNAL processes.

7.2.1.1 CLOCK acting on generators.

This map is defined on generators as indicated in the following table.

Table 5. The image by CLOCK of the generators

generator	its image by CLOCK
non boolean function F	F **cut** !F
boolean function	unchanged
q := $p	(q := $p) **cut** q
b := a **when** h	unchanged
x := a **default** b	unchanged
h := tt(c)	unchanged

In this table, "boolean functions" denote SIGNAL functions built upon the boolean algebra *and, or, not*, while "non boolean functions" denote the other SIGNAL functions; in particular, primitive boolean expressions are non boolean functions according to this definition. The shortage F **cut** !F indicates that all output ports of F are cut.

7.2.1.2 Complete definition of the map CLOCK.

Let us assume that

$$CLOCK(P) = P\,\mathbf{cut}\,p_1,\ldots,p_k$$
$$CLOCK(Q) = Q\,\mathbf{cut}\,q_1,\ldots,q_n$$

a property which is satisfied for the generators. Then the following formulas provide us by induction with the complete definition of the map CLOCK.

$$CLOCK(P\ ?a{:}b) = CLOCK(P)\ ?a{:}b \tag{7-7}$$

$$a \notin \{p_1,\ldots,p_k\}:\ CLOCK(P!a{:}b) = CLOCK(P)\ !a{:}b \quad (i)$$
$$CLOCK(P\ !p_1{:}b) = CLOCK(P)\ ?*p_1{:}*b \quad (ii) \tag{7-8}$$

$$CLOCK(P\&Q) = CLOCK(P)\&CLOCK(Q) \tag{7-9}$$

$$b \notin ?P \cap !P:\ CLOCK(P@b) = CLOCK(P) \quad (i)$$

$$b \in ?P \cap !P, b \notin \{p_1, \ldots, p_k\}: \quad CLOCK(P@b) = CLOCK(P)@b \quad (ii)$$
$$CLOCK(P@p_1) = CLOCK(P)\ ?p_1{:}*p_1 \quad (iii) \qquad (7\text{-}10)$$

Then, the formulas (7-3,7-4,7-5, 7-6) show by induction that

LEMMA 7: For every SIGNAL process P, CLOCK(P) is of the form

$$CLOCK(P) = P\mathrm{cut}\, p_1, \ldots, p_k \qquad (7\text{-}11)$$

for some subset $\{p_1, \ldots, p_k\}$ of the output ports of P. □

As a consequence, the lemmas 2 to 6 can be applied with Q = CLOCK(P). The sequel of the chapter will be devoted to show that CLOCK(P) is indeed equivalent to the clock calculus of P.

7.2.2 Relating CLOCK(P) and the clock calculus of P.

7.2.2.1 Deleting the memories of CLOCK(P).

First, the definition of CLOCK(q := $p) shows clearly that the memory of the resulting process play no role in its running: none of the properties we want to investigate relies on the memory status. As a consequence, we can delete the memory in the image by CLOCK of the generator $. By induction, the memories of the image by CLOCK of any SIGNAL process can be deleted, without modifying the timing properties of this image.

7.2.2.2 Restricting the data types.

Since non boolean functions have all their output ports cut, non boolean values play no role in the timing properties of the images by CLOCK of the generators. The same holds by induction for any SIGNAL process. Finally, since non boolean values are carried by input ports only in the image by CLOCK of any process, we can substitute the single type *dummy* (see the chapter on the clock calculus) to any non boolean type. *We shall denote by CLOCK* the map we have defined on SIGNAL by removing the memories and restricting the non boolean types to the dummy type for every process of the form CLOCK(P).*

7.2.2.3 CLOCK*(P) is identical to the clock calculus of P.

This claim is easily verified on the generators. To show that it is true for every process P, we have to show that the transfer formulas of the clock calculus (see the table 2) are equivalent to the transfer formulas of the map CLOCK we have shown above. To show this, the only fact we have to pinpoint is that the marker ✱ inserted in front of the cut output ports of the process Q in the definition 6 corresponds to the marker ! used in the transfer formulas of the clock calculus. The complete correspondance between transfer formulas is shown in the following table, where a,b refer to ports which are not cut, whereas p refer to a cut port of P:

Table 6. The correspondance between clock calculus and CLOCK transfers

CLOCK transfer	clock calculus transfer
CLOCK(P ?a:b) = CLOCK(P) ?a:b	EQ(P ?a:b) = EQ(P) ?a:?b
CLOCK(P !a:b) = CLOCK(P) !a:b CLOCK(P !p:b) = CLOCK(P) ?✱p:✱b	EQ(P !a:b) = EQ(P) !a:!b EQ(P !p:b) = EQ(P) !p:!b
CLOCK(P & Q) = CLOCK(P) & CLOCK(Q)	EQ(P & Q) = EQ(P) U EQ(Q)
CLOCK(P @ b) = CLOCK(P) @ b CLOCK(P @ p) = CLOCK(P) ?p:✱p	EQ(P @ b) = EQ(P) ?b:!b EQ(P @ p) = EQ(P) ?p:!p

Finally, we have completely proved that the clock calculus of a process is entirely equivalent, as far as timing is concerned, to the result of cuts made on this process. This will allow us to apply to the clock calculus the lemmas 2 to 6 above.

7.3 The clock calculus as a tool to check timing properties of a SIGNAL process.

- In the sequel, we shall say that the clock calculus of a process P is *solvable* if no clock of P is constrained to be void .
- We shall say that a solved clock calculus does not involve any *internal* phantom if every clock can be expressed as the result of a polynomial function of the primitive boolean expressions and the input ports, but no extra phantom.[6]

---

6 the example 2 exhibit a phantom, but no internal phantom, whereas the examples 1 and 5 exhibit internal phantoms.

- We shall say that a solved clock calculus does not involve any *i – o – internal* phantom if every clock can bve expressed as the result of a polynomial function of the primitive boolean expressions, the input ports, and the output ports, but no extra phantom.[7]

MAIN THEOREM: Let P be a SIGNAL process.

1/ If $\pi = \{p_1, ..., p_k\}$ *is a time – incorrect set of input ports of P, then the clock calculus yields* $p_1 = ... = p_k = 0$.

2/ If the clock calculus of P is solvable, and if the conditional dependence graph of P is circuit – free[8]*, then P is fully time – correct and all its transitions are computable.*

3/ If, furthermore, the solution of the clock calculus of P does not involve internal phantoms, then P is deterministic; if the solution of the clock calculus of P does not involve i – o – internal phantom, then P is weakly deterministic. ■

PROOF: Let us first prove 1/. This is a direct consequence of the lemma 3, since the property of time – incorrectness for a set of signals of a SIG process is obviously equivalent to the property that these signals be constrained to be zero in the solved clock calculus. 2/ has already been proved. The assumption in 3/ is equivalent to assume that every acceptance of the considered SIG process can be used by a single transition of this SIG process, so that 3/ is a direct consequence of the lemma (6 – i); the second assertion of 3/ is proved in the same way using lemma (6 – ii).

CONCLUSION OF THE CHAPTER: The main theorem has been proved via the technique of cuts. But this technique can also be applied to other "clock calculi", since the choice of cuts we have made for the generators was rather arbitrary. This choice was motivated by the following arguments:

- we have canceled the effect of all non boolean values; in fact, non boolean values play a role only through primitive boolean expressions, and we chosed to refuse to reason about such expressions;
- we have canceled the memories, to get finally a "purely static" process as result of the cuts: this was a drastic way to eliminate any risk of a need for non terminating simulations to check the timing of cut processes.

---

7 the examples 1 and 5 exhibit no i – o – internal phantoms.

8 in the sense of the theorem 4

The first choice is certainly a must. However, the second one could be refined: some dynamics could be kept in the cut process, provided we are sure that checking the timing characteristics of this process does not require non terminating simulations. This is for example the case if the cut process is a finite state periodic automaton, which is for example the case for the automata associated to periodic counters; such a situation is discussed in the analysis of the *time-multiplexer* in the last chapter.

Note that extending the use of the temporal logic [Pnuelli 1977] [Lamport 1985] to the study of SIGNAL programs would have been non trivial because of the effect of the external stimuli. A technique similar to cuts should have been used also, since SIGNAL processes are not pure synchronisation processes. The comparison will be even clearer for the dynamical clock calculus we shall outline in the chapter 9.

Chapter Eight
THE COMPUTATIONAL SEMANTICS.

This part of the semantics specifies how the SIGNAL programs are executed; this will be achieved by further refining the behavioral semantics we have introduced in the chapter 3. We shall give the computational semantics of fully time-correct processes only. We shall give two different kinds of computational semantics, namely a data-flow oriented semantics, and a sequential one in the style of [Berry & Cosserat, 1984]. The reason for giving a data-flow oriented computational semantics is that the corresponding execution scheme will exhibit the maximum amount of parallelism, while the sequential semantics will be purely sequential.

8.1 The data-flow computational semantics.

It will be based upon the notion of *data-flow graph* we shall introduce now. Data-flow graphs will be generically denoted by DFG.

8.1.1 Data-flow graphs.

Given a process P, its data-flow graph is defined as follows.

STEP 1; let the solved clock calculus EQ(P) be a set of expressions of the form

$$X = \Pi(\beta_1, \ldots, \beta_n, \chi_1, \ldots, \chi_m, \Phi_1, \ldots, \Phi_k) \tag{8-1}$$

where $\{\beta_1, \ldots, \beta_n\}$ are free input clocks, $\{\chi_1, \ldots, \chi_m\}$ are primitive boolean expressions, and $\{\Phi_1, \ldots, \Phi_k\}$ are internal phantoms[9]. We removed external phantoms, just assuming that the free input clocks match the constraints resulting from the clock calculus. Free clocks of EQ(P) will be generically denoted by κ.

STEP 2: Then, further label the branches of the conditional dependence graph by the generator which caused the considered dependency. This gives a graph whose branches are of the form

---

9 see the main theorem

$$p \xrightarrow{\text{gen}\ ;\ h} q \tag{8-2}$$

where p, h, q are as before in DG(P), and gen is the generator which caused the considered dependency. The result is denoted by DGG(P).

STEP 3: subgraphs of DFG(P) are obtained according to the following rules.

Rule of the input ports. Let a be an input port of P.

$$\frac{DGG(P) \ni a \rightarrow \quad EQ(P) \ni a = \Pi(\kappa_1 \ldots \kappa_p)}{DFG(P) \ni \xrightarrow{\kappa_i} \Pi \xrightarrow{\text{clock}(a)}}$$

We want to distinguish between a non boolean signal and its clock, hence this notation. In this formula, Π is assumed to be nontrivial; consequently, since external phantoms have been removed, at least one of the κ_i's must be a primitive boolean expression.

Rule of the delay.

$$\frac{DGG(P) \ni o\$x \xrightarrow{y:=\$x\ ;\ h} y\ ;\ x \xrightarrow{y:=\$x\ ;\ h} n\$x \quad EQ(P) \ni h = \Pi(\kappa_1 \ldots \kappa_p)}{DFG(P) \ni \xrightarrow{\kappa_i} \Pi \xrightarrow{\text{clock}(x)} [y := \$x] \xrightarrow{y} ;\ \xrightarrow{x} [y := \$x] \xrightarrow{n\$x}}$$

Rule of the when.

$$\frac{DGG(P) \ni p \xrightarrow{q:=p\text{when}k\ ;\ h} q \quad EQ(P) \ni h = \Pi(\kappa_1 \ldots \kappa_p)}{DFG(P) \ni \xrightarrow{p} [q := p\text{when}h] \xrightarrow{q} ;\ \xrightarrow{\kappa_i} \Pi \xrightarrow{h} [q := p\text{when}h] \xrightarrow{q}}$$

Thhesignal k can be equivalently replaced by h, since the clock calculus indicates that the result of this generator is the same in both cases.

Rule of the other generators, except tt.

$$\frac{DGG(P) \ni \; p \xrightarrow{\text{gen} \; ; \; h} q}{DFG(P) \ni \; \underline{p} \rightarrow \; [\text{gen}] \; \underline{q} \rightarrow}$$

There is no rule for the generator tt, since its effect is fully taken into account by the clock calculus. In these rules, all the polynomial functions coming from the solved clock calculus can be squared to get the form

$$h^2 = \Omega(\kappa_1, \ldots, \kappa_p) \tag{8-3}$$

since the value taken by the result in the set $\{-1,1\}$ play no role in the synchronisation mechanisms of the rules we have introduced.

STEP 4: Finally, DFG(P) is obtained by interconnecting the above introduced subgraphs via the identity of the labels of the branches, keeping in mind that we must distinguish between a non boolean signal and its clock.

8.1.1.1 Data-flow graphs of some examples.

The data-flow graphs of the examples 8 and 10 are shown below.

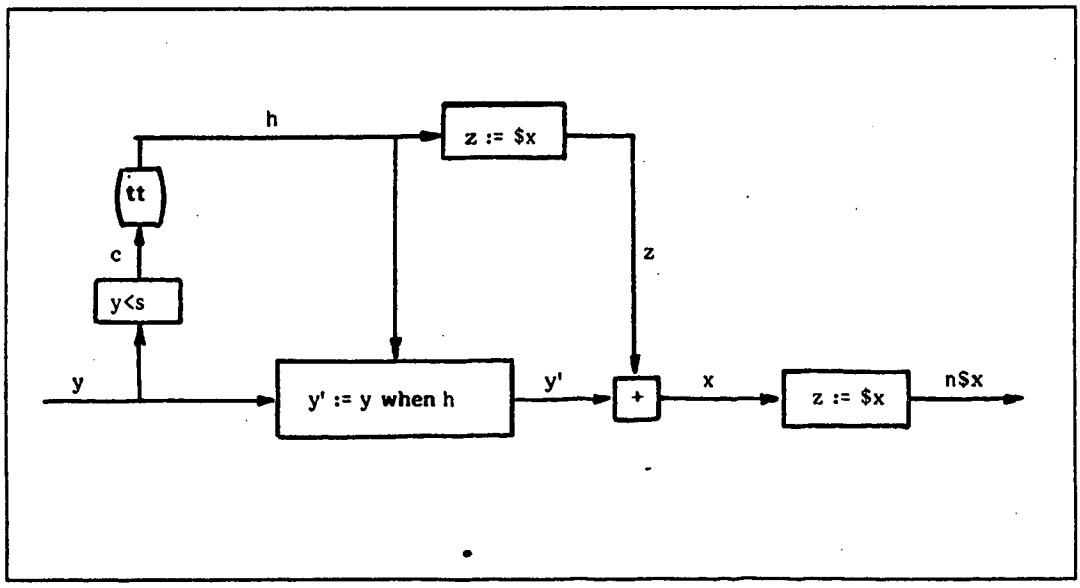

Figure 6. Data-flow graph of the example 8

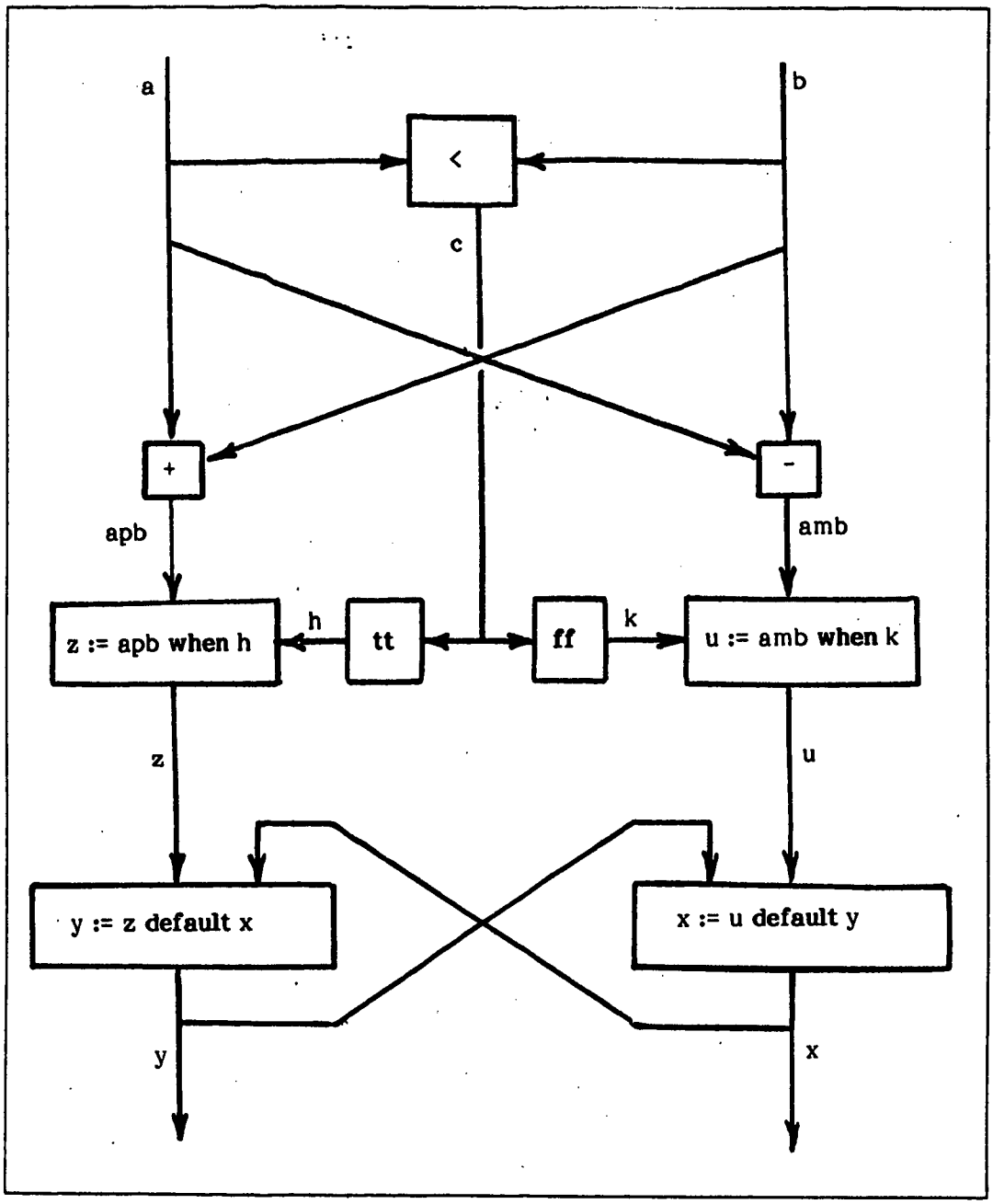

Figure 7. Data-flow graph of the example 10

8.1.2 The computational semantics of data-flow graphs.

8.1.2.1 Semantics of the nodes.

The semantics of a node will be given as follows

- associate to each node a list of transitions we shall denote by TT;
- defined the data-flow firing mode associated to the corresponding transition.

There are three types of nodes

1. the generator $,
2. other SIGNAL generators,
3. equations of the clock calculus of the form (8-3).

The transitions of the delay.

$$\frac{DFG(P) \ni \xrightarrow{clock(x)} [y := \$x] \xrightarrow{y} ; \xrightarrow{x} [y := \$x] \xrightarrow{n\$x}}{TT(y := \$x): \ < x[u] > \xrightarrow[y[u]]{clock(x)} ; \xrightarrow{x[v]} < x[v] >}$$

The corresponding data-flow firing mode is depicted in the figure 8 below:

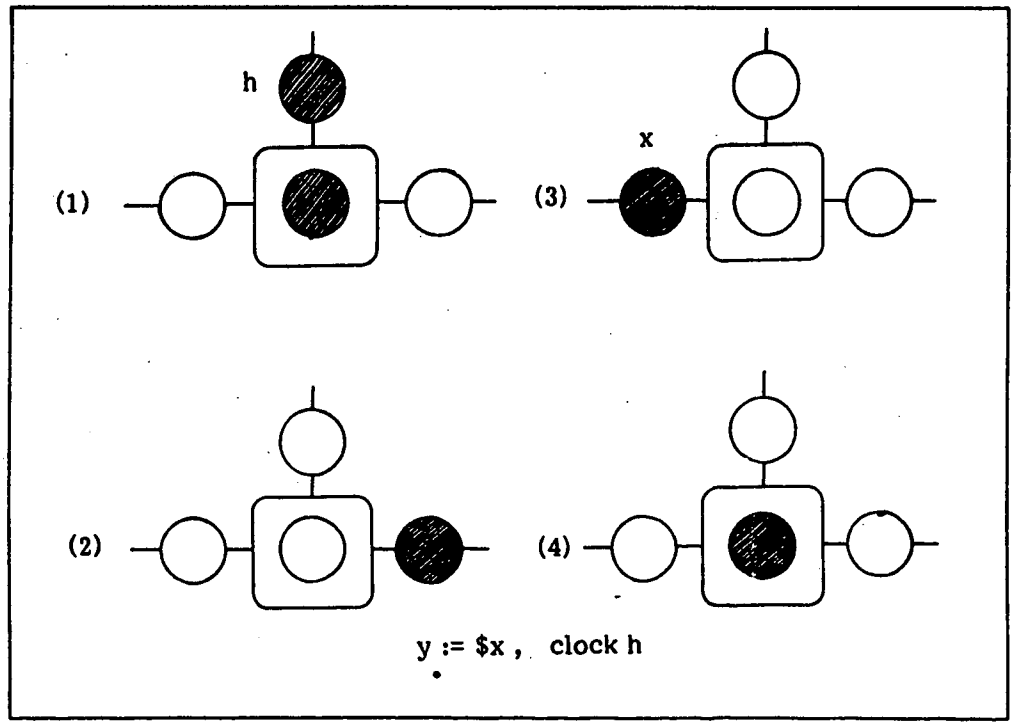

Figure 8. Data-flow firing mode of the delay

The token inside the rectangle denotes the availability of values in the memory. The diagrams (1) and (2) show the memory fetch, while the diagrams (3) and (4) show the writing of the incoming data inside the memory at the end of the transition.

The transitions of the generators. Let gen be a generator involved in the process P.

$$\frac{T(\text{gen}):\ <\$> \xrightarrow[f]{a[x]\ b[.]\ldots} <\$'> \quad \xrightarrow{\text{clock}(a)} \text{gen} \in DFG(P)\ ;\ \xrightarrow{\text{clock}(b)} \text{gen} \notin DFG(P)}{TT(\text{gen};P):\ <\$> \xrightarrow[f]{\text{clock}(a)[1]\ a[x]\ b[.]\ldots} <\$'>}$$

$$\frac{T(\text{gen}):\ <\$> \xrightarrow[f]{a[\perp]\ b[.]\ldots} <\$'> \quad \xrightarrow{\text{clock}(a)} \text{gen} \in DFG(P)\ ;\ \xrightarrow{\text{clock}(b)} \text{gen} \notin DFG(P)}{TT(\text{gen};P):\ <\$> \xrightarrow[f]{\text{clock}(a)[0]\ a[\perp]\ b[.]\ldots} <\$'>}$$

The corresponding data - flow firing mode is depicted in the figure 9 below

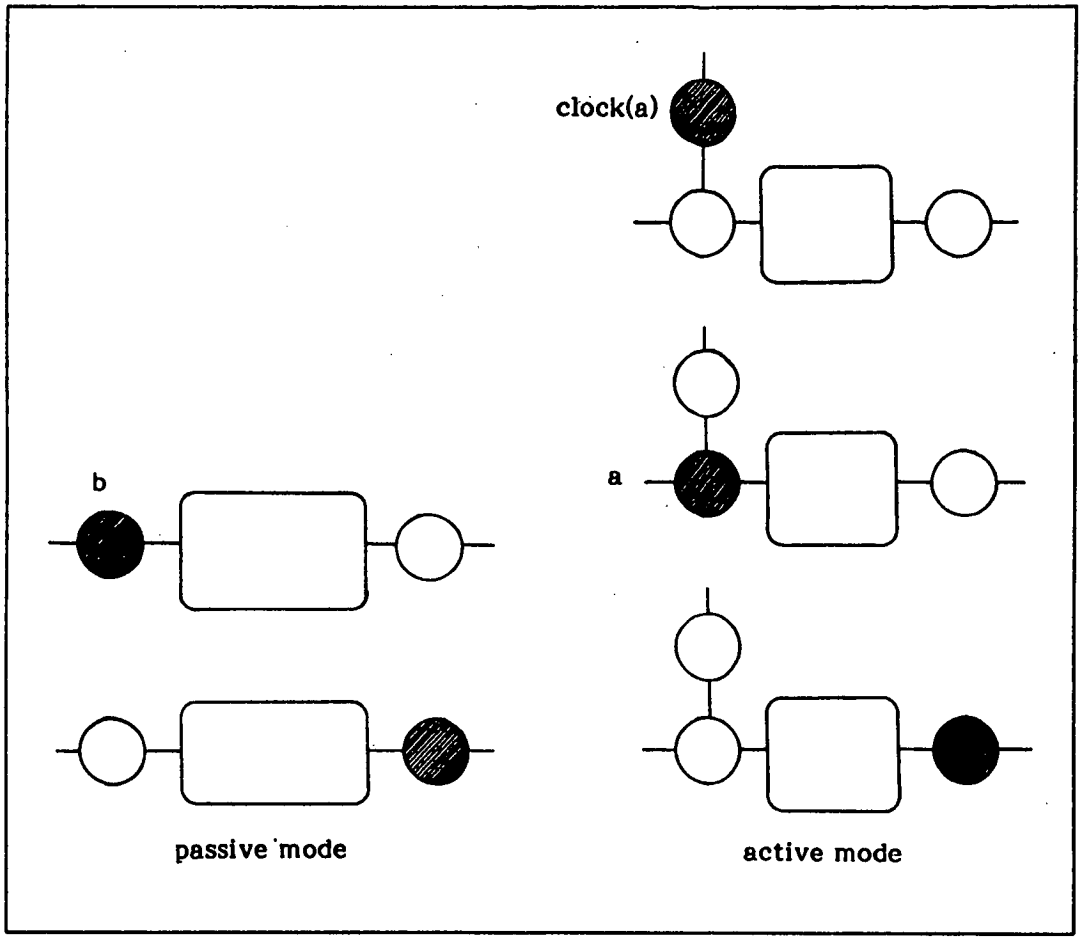

Figure 9. Data-flow firing mode of the generators

This figure shows that the input port *b* acts according to the *data-driven* mode, since the availability of data is a perequisite for firing. On the other hand, the port *a* acts according to the *demand-driven* mode, since clock(*a*) acts exactly as a token to request the value carried by *a*.

The transitions of the clock equations.

$$\frac{EQ \ni h^2 = \Omega(\kappa_1 \ldots \kappa_p)}{TT(\Omega; P): \quad <.> \frac{\kappa_1[\xi_1] \ldots \kappa_p[\xi_p]}{h[\theta := \Omega(\xi_1 \ldots \xi_p)]} \rightarrow <.>}$$

In other words, the clock equations act as token generators

8.1.2.2 The semantics of data - flow graphs.

This is just the usual semantics of data - flow graphs (see [Dennis 1974]) in the mixed *data/demand - driven* mode of computing we have described; if we furthermore assume that neither actions nor communications take time, we know that *the execution of the program requires no file at the input ports.* The results of the proceding chapters ensure that this computational semantics is a refinment of the behavioral semantics of the chapter 3 firing of the considered process, consistent with the usual computational semantics of our instantaneous language we have introduced in the chapter 2.

8.2 The sequential computational semantics.

The principles of the sequential semantics are different. The idea is the following: we have at our disposal only a single "processor" to fire successively all the elementary actions involved in a transition. The key tools will be again the conditional dependence graph, together with the clock calculus, but they will be used in a different way. We shall construct an automaton, called the *graph automaton*, GA(P) for short, which will cause the successive firing of the elementary actions in a consistent way.

As for the data - flow semantics, the objects the graph automaton will be based upon is the pair $\{DG(P), EQ(P)\}$ of the conditional dependence graph and the solved clock calculus formulated as in (8-1).

8.2.1 The domains of GA(P).

8.2.1.1 The states of GA(P).

Let us introduce the following objects.

Determinate clocks.

H : the set of the clocks which label the branches of DG(P);

$$H^{\top} \subset H,\ H^{\perp} \subset H:\ H^{\top} \cap H^{\perp} = \phi$$

The set $H^{\top}$ has to be interpreted as the subset of the clocks that are known to be 1 (in the clock algebra) and $H^{\perp}$ as the subset of the clocks that are known to be 0 at the considered stage of the execution of the transition.

Nested dependence graphs.

The set of the subgraphs of DG(P) is endowed with the following partial order

$$D' \subseteq D$$

$$\Leftrightarrow$$

$$\left\{ \begin{array}{l} [x \xrightarrow{k} a] \in D \quad and \quad k \subseteq h \\ [a \xrightarrow{h} b] \in D; \notin D' \end{array} \right\} \Rightarrow \left\{ [x \xrightarrow{k} a] \notin D' \right\} \tag{8-4}$$

Note that this partial order can be directly derived from the level of the nodes in DG(P) when this graph is acyclic, but we have seen that computable conditional dependence graphs can still exhibit cycles, so that another technique must be used to define this partial order.

The states of GA(P) is the set of the triples

$$< D, H^{T}, H^{\perp} > \tag{8-5}$$

such that

$$\{ [a \xrightarrow{h} z] \in D \} \Rightarrow \{ h \notin H^{\perp} \} \tag{8-6}$$

i.e., the branches of D that are already known to be not involved in the considered transition of P are deleted.

8.2.1.2 The outputs of GA(P).

Elementary transitions of P.

DEFINITION: Let y := $x be a generator involved in P as a subprocess; then

$$< \$x[u] > \xrightarrow[y[u]]{} \bullet$$

$$\xrightarrow[x[v]]{} < \$x[v] >$$

are elementary transitions of P. On the other hand, all the transitions of the generators involved in P which are not delays are also elementary transitions of P. Elementary transitions are denoted by ET. *The outputs of GA(P) are the elementary transitions of P.* A given output indicates which transition of which generator is fired at a given stage of the execution of a transition of P.

8.2.1.3 The inputs of GA(P).

Inputs of GA(P) are t-uples of the form

$$bexp_1[\beta_1],\ldots,bexp_t[\beta_t] \tag{8-7}$$

where *bexp* denote primitive boolean expressions, and β the boolean value they carry. Intuitively, inputs of GA(P) are the result of the evaluation of primitive boolean expressions during the execution of the considered transition. In fact, these inputs will be the byproduct of the firing of the elementary transition which is the output of GA(P).

8.2.2 The transitions of GA(P).

Transitions of GA(P) are written as follows

$$< D, H^{\top}, H^{\perp} > \frac{bexp_1[\beta_1] \ \ldots \ bexp_t[\beta_t]}{ET} \rightarrow < D', H'^{\top}, H'^{\perp} > \tag{8-8}$$

For (8-8) to be a transition of GA(P), the input, output, and new state of GA(P) must be as follows.

8.2.2.1 Defining the output ET.

The nodes of DG(P) which don't belong to D are those that have been already evaluated, and that will no more be used in the sequel of the execution. The source nodes of D are those that have been evaluated, and that will be further needed in the sequel.

$H^{\top}$ is the subset of H containing the clocks that are already *known* to be 1, whereas $H^{\perp}$ is the subset of H containing the clocks that are already known to be 0.

An elementary transition ET of P is *ready to be fired* if every branch

$$a \xrightarrow{h} z$$

of DG(P) caused by this elementary transition is such that

$$\{a \text{ is a source node of } D\} \text{ and } \{h \in H^{T}\}$$
$$\text{or}$$
$$\{h \in H^{\perp}\} \qquad (8\text{-}9)$$

Assume a rule has been given to select one among the transitions that are ready to be fired, the output of GA(P) is then the considered elementary transition. Note that the output depends only on the current state of GA(P), and not on its input.

8.2.2.2 Defining the input.

The input of the considered transition of GA(P) is the t-uple (8-7) produced by the output of the considered transition of GA(P), i.e. is the result of the firing of the selected elementary transition of P. This t-uple must be considered as an input of GA(P), since it is the result of the firing, but not a part of the firing itself.

8.2.2.3 Defining the new state.

Knowing the previous state, the output, and the input of GA(P), we can compute

$$H'^{T} \supset H^{T}, \quad H'^{\perp} \supset H^{\perp}$$

which are the new sets of clocks of value 1 and 0 respectively. These sets are growing when new primitive boolean expressions are evaluated, which enter in instructions such as **tt**.

The new subgraph D' is then built as follows.

1. Delete the branches corresponding to the elementary transition which has been fired; delete also the possibly created isolated nodes.
2. Delete the branches labelled with clocks belonging to $H'^{\perp}$.

8.2.3 Running GA(P).

Let $o\mu$ be the initial state of P before the considered transition.

Let $\{\beta_1,\ldots,\beta_n\}$ be the free input clocks as in (8-1); if $\beta_i \neq 0$, then the corresponding input port carries an effective value, whereas it carries $\perp$ in the other case.

Candidates effective values are assumed to be available to the other input ports, and will be used or not according to the values of the primitive boolean expressions resulting from the running of GA(P).

Choose arbitrarily the values of the internal phantoms involved in (8-1). This allows us to compute, using EQ(P), the sets

$$H_0^{\top} \text{ and } H_0^{\perp}$$

Finally, delete in DG(P) the branches which are labelled with elements of $H_0^{\perp}$ to get D_0.

Transitions (8-8) of GA(P) are successively applied, yielding a sequence of states

$$< D_0, H_0^{\top}, H_0^{\perp} > \rightarrow < D_1, H_1^{\top}, H_1^{\perp} > \rightarrow \ldots$$

THEOREM 5 : Let # be the cardinal of the set of the generators involved in P, and #$ be the cardinal of the generators "delay"; set N = # + #$. Then we have

$$D_N = \phi$$
$$H_N^{\top} \cup H_N^{\perp} = H \quad \square$$

PROOF: since the process is time-correct, the cycles taking place in DG(P) are broken during the running of the graph automaton. Since no other cause of starvation can occur, the run of GA(P) terminates at most after N transitions, since a delay generator splits into two elementary transitions.

8.2.4 Examples.

The running of GA(P) is illustrated on the examples 8 and 10. In these figures, the clocks which are written in boldface and underlined denote the elements of $H^{\top}$, while the other clocks are not evaluated at the considered stage of the algorithm.

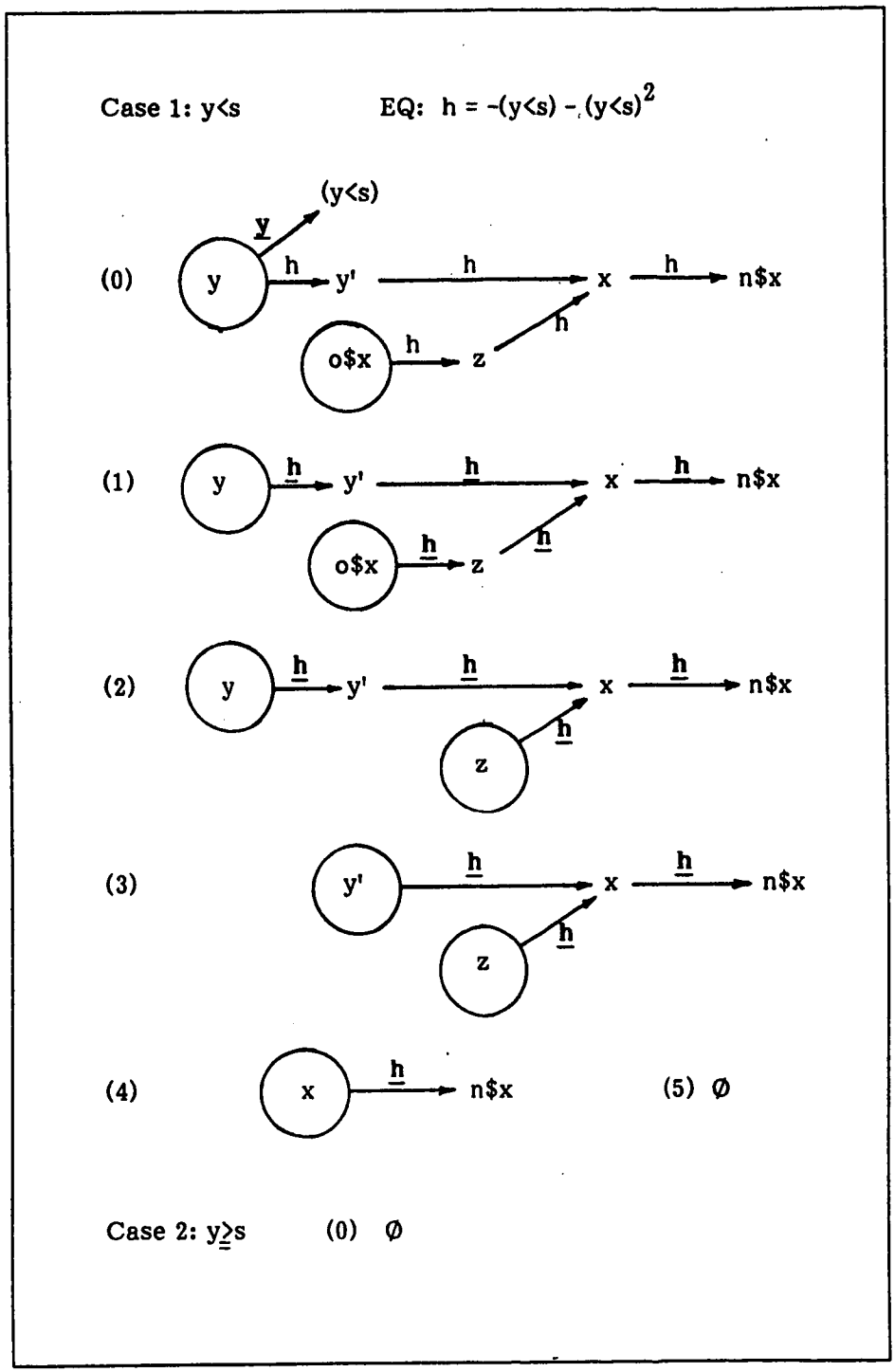

Figure 10. running *GA*[P] on the example 8

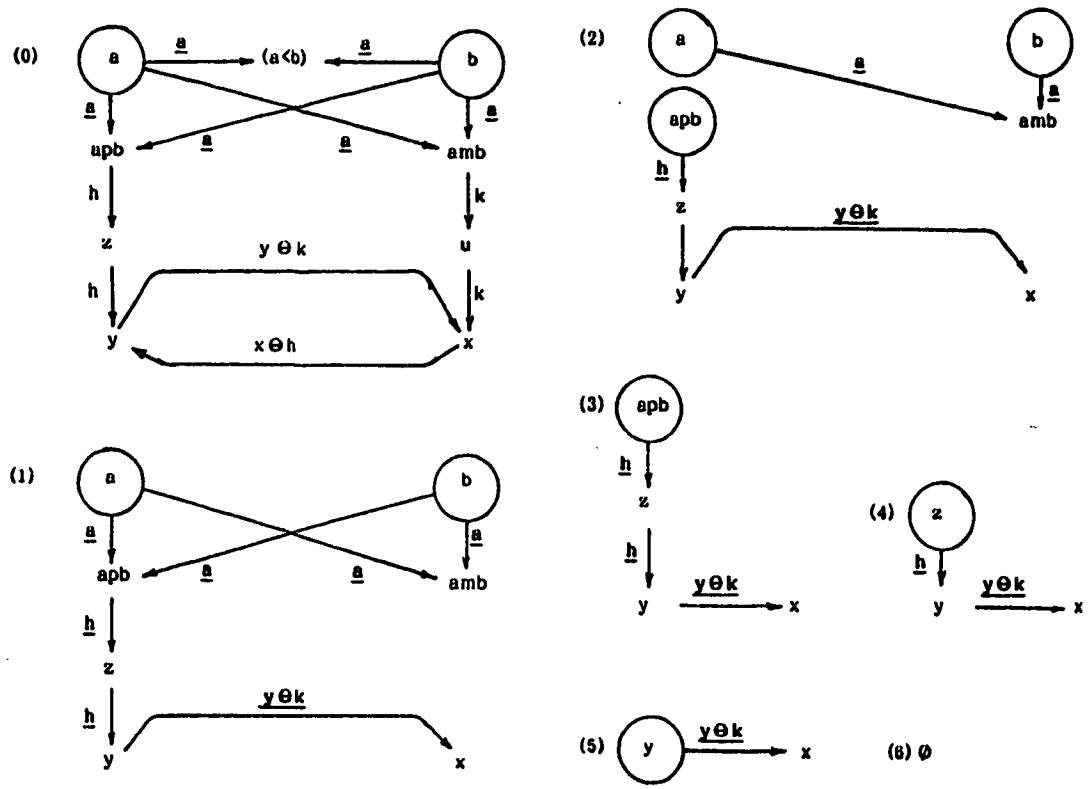

Figure 11. running *GA*[P] on the example 10

8.2.5 Separate compilation.

The graph automaton is not suitable to separate compilation. The reason is that the environment of a process generally influences the solution of its clock calculus (bindings of free clocks can be caused by the environment) and the partial order introduced on its conditional dependence graph. This phenomenon was already pointed out in [Berry & Cosserat 1984]. However, the pair {solved clock calculus, conditional dependence graph} is the level of compilation which is suitable for

separate compilation: simple rules are required to link such pairs corresponding to interconnected subprocesses, and on the other hand, both computational semantics are easily derived from the solved clock calculus and the dependence graph. Finally, some subprocesses of a correct SIGNAL process possess a graph automaton which is independent of the environment: such processes are currently studied by Le Goff & Besnard [in preparation], and are called «clusters».

Chapter Nine
A MORE SUBTLE EXAMPLE.

The time multiplexing of signals is roughly described as follows. Given an ordered n-uple $x_1 \ldots x_n$ of signals, the time multiplexer delivers the singleton:

$$\ldots\ x_1(t)\ x_2(t)\ \ldots\ x_n(t)\ x_1(t+1)\ x_2(t+1)\ \ldots \tag{9-1}$$

To simplify, we shall restrict ourselves to the case where all the x_i's have the same clock. This example will illustrate the capabilities of the language SIGNAL to generate clock *oversampling*, i.e. to add new events to the input events. At the same time, its analysis will be a good opportunity to introduce the reader to interesting extensions of our clock calculus.

9.1 The time-multiplexer: the program.

To avoid to introduce the *loops* (which is not our purpose here), we shall restrict ourselves to $n = 3$. The following syntax will be used to introduce hierarchy in the language (see [Le Guernic & al. 1986] for more details on the complete syntax of the language SIGNAL):

```
PROCESS { ? typed listin ! typed listout } =

        body

        where

        SUBPROCESS1 { ... }

        ...

        SUBPROCESSk { ... }

end PROCESS
```

The subprocesses described in the block **where** follow the same syntax, and are used in the body of the process. When no type is mentioned, any type is allowed. We shall make use of the SIGNAL function **synchro**, which has only inputs and no outputs, so that the effect of this function

is a pure synchronization of its inputs. We shall also use the operator !" acting on output ports: the instruction

```
P !! list
```

indicates that all the output ports that don't belong to "list" are *masked*, i.e. receive a private name, which cannot be used by the programmer . According to this syntax, let us propose the following program, when the input signals are known to have the same clock:

```
MUXSYNCHRO { ? x1, x2, x3   !y } =

        ( (COUNTMOD ? x:x1) &

        (VAR ? s:x2, h:n  ! u:x'2) & (x"2 := x'2 when tt(n=2)) &

        (VAR ? s:x3, h:n  ! u:x'3) & (x"3 := x'3 when tt(n=3)) &

        (y := (x1 default x"2) default x"3) )

        @ n,x'2,x'3,x"2,x"3   !!y

       where

       VAR { ?s,h ! u } =

              ((u := s default zu) & (zu := $u) & synchro(u,h))

               @ u,zu  !! u

       end VAR

       COUNTMOD { ? x  ! int n } =

              ((n := zn + 1) & (zn := (0 when x) default zzn) & (zzn := $n) &

              (k := tt(zzn = 3)) & synchro(x,k))   @ n, zn, zzn, k   !! n

       end COUNTMOD

end MUXSYNCHRO
```

COMMENTS: The process COUNTMOD is a counter with reset: the integer n counts the occurrences of the signal h with a reset to the value 1 when x occurs. This value must be

accepted when the previous value zzn of n is equal to 3. The process VAR is a "synchronized variable": it delivers the current value, and if not available the past value of the input signal s in synchrony with h .

9.2 The clock calculus of the MUXSYNCHRO.

We shall illustrate how the separate resolution of clock calculi is done.

9.2.1 Clock calculus of COUNTMOD.

The primitive clock calculus is the following, using the notations of (6–3), and writing c for short instead of [zzn = 3]:

$$\begin{aligned} n \equiv zn \equiv zzn \equiv c \quad & (i) \\ zn \equiv x \oplus zzn \quad & (ii) \\ k = -c - c^2 \quad & (iii) \\ x \equiv k \quad & (iv) \end{aligned} \tag{9-2}$$

Its solution is

$$\begin{aligned} n \equiv c \quad & (i) \\ k = -c - c^2 \quad & (ii) \\ x \equiv k \quad & (iii) \end{aligned} \tag{9-3}$$

where the boolean c is the free parameter.

9.2.2 Clock calculus of VAR

The primitive clock calculus is

$$\begin{aligned} u \equiv s \oplus zu \\ zu \equiv u \equiv h \end{aligned} \tag{9-4}$$

which is solved as

$$s \equiv \Psi \otimes h \quad (i)$$
$$u \equiv h \quad (ii) \tag{9-5}$$

where Ψ is a phantom.

9.2.3 The clock calculus of MUXSYNCHRO

The primitive clock calculus is, using (9-3, 9-5), and denoting by x for short the clock of the input ports, and by $c2$ and $c3$ respectively the primitive boolean expressions [n=2] and [n=3]:

$$\begin{gathered}
x1 \equiv x2 \equiv x3 \equiv x \\
n \equiv c \equiv c2 \equiv c3 \\
k = -c - c^2 \\
x \equiv k \\
x \equiv y \otimes \Psi \\
x'2 \equiv x'3 \equiv n \\
h2 = -c2 - c2^2 \\
h3 = -c3 - c3^2 \\
x''2 \equiv x'2 \otimes h2 \\
x''3 \equiv x'3 \otimes h3 \\
y \equiv x1 \oplus x''2 \oplus x''3
\end{gathered} \tag{9-6}$$

The solution of this calculus is immediate, and is given by

$$\begin{gathered}
c \equiv c2 \equiv c3 \\
k = -c - c^2 \\
h2 = -c2 - c2^2 \\
h3 = -c3 - c3^2 \\
x \equiv k \\
y \equiv k \oplus (c \otimes (h2 \oplus h3))
\end{gathered} \tag{9-7}$$

In this calculus, we have deleted the signals which are not visible outside, and which play no role in the solution of the calculus.

Note that , in this solved calculus, the free parameters are the three primitive boolean expressions c, c2, c3, while the input is constrained.

9.3 The conditional dependence graph.

9.3.1 The conditional dependence graph of COUNTMOD.

It is shown in the figure 12 below

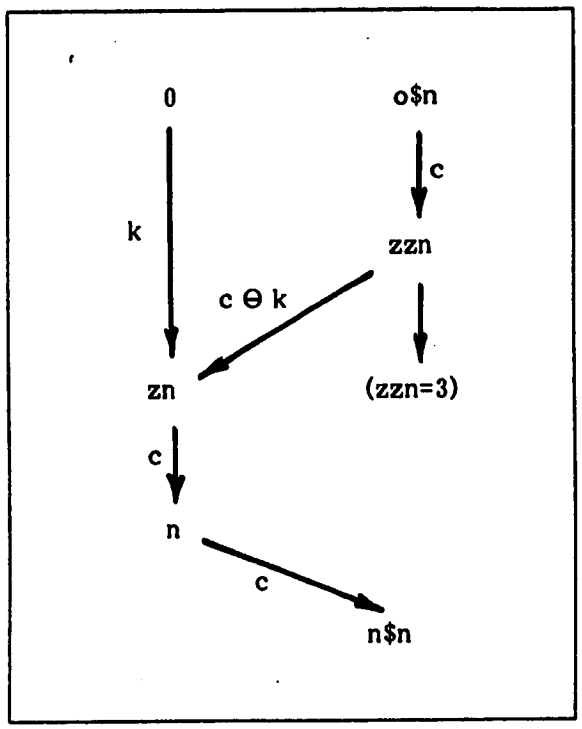

Figure 12. Dependence graph of COUNTMOD

9.3.2 The conditional dependence graph of VAR.

It is shown in the figure 13 below

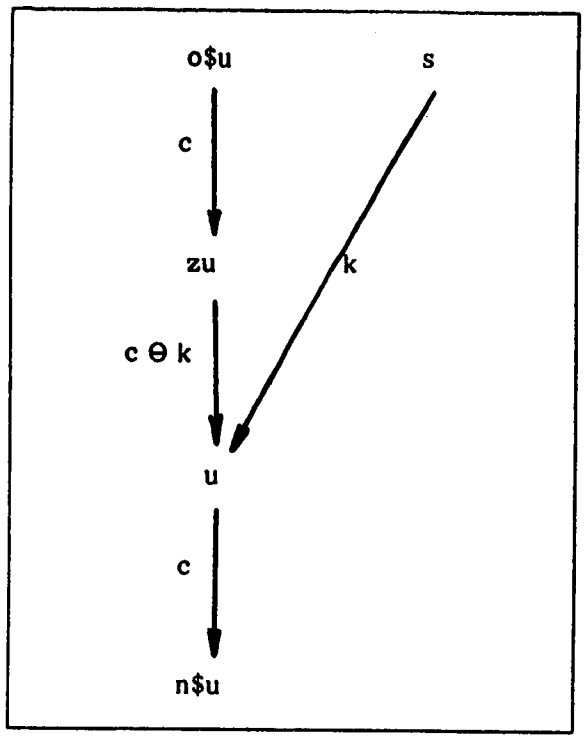

Figure 13. Dependence graph of VAR

9.3.3 The conditional dependence graph of MUXSYNCHRO.

It is shown in the figure 14 below

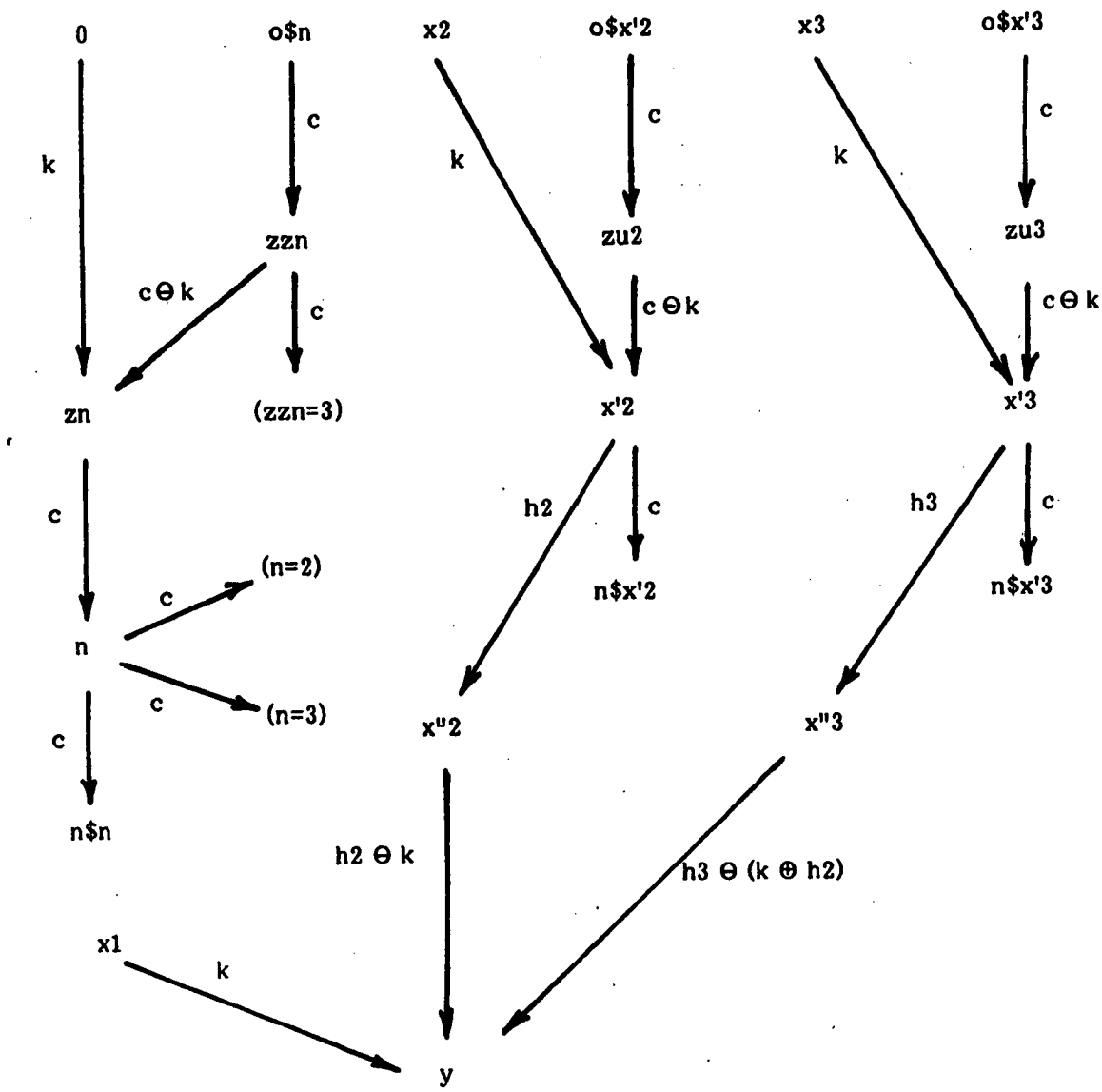

Figure 14. Dependence graph of MUXSYNCHRO

CONCLUSION: This graph is circuit-free, so that, since the clock calculus has been solved without the need of any phantom, *we have proved that MUXSYNCHRO is fully time-correct and deterministic.* In other words, we have been able to oversample input clocks,

while keeping able to fully control the timing of this program. This illustrates the power of the clock calculus. However, this calculus is not able to prove completely the program, in the sense that it does not ensure that the three components are produced in the wished circular ordering. This is a point we shall now further investigate.

9.4 An introduction to the dynamical clock calculus.

Let us modify the program as follows

```
MUXSYNCHRODYN { ? x1, x2, x3   !y } =

        ( (COUNTMOD ? x:x1) &

        (VAR ? s:x2, h:n  ! u:x'2) & (x"2 := x'2 when c2) &

        (VAR ? s:x3, h:n  ! u:x'3) & (x"3 := x'3 when c3) &

        (y := (x1 default x"2) default x"3) )

        @ n,x'2,x'3,x"2,x"3   !!y

        where

        VAR { ?s,h ! u } =

                ((u := s default zu) & (zu := $u) & synchro(u,h))

                 @ u,zu  !! u

        end VAR

        COUNTMOD { ? x  ! bool c1, c2, c3 } =

                ((c2 := $c1) & (c3 := $c2) & (c1 := $c3) &

                (k := tt(c1 and not c2 and not c3)) & synchro(x,k))

                @ c1 , c2 , k   !! c1, c2, c3

        end COUNTMOD
```

end MUXSYNCHRODYN

We shall here analyse the timing of this program; the key point is the analysis of COUNTMOD.

9.4.1 The clock calculus of MUXSYNCHRODYN.

The analysis of COUNTMOD will suffice for our purpose. Using the clock calculi of the boolean expressions, this clock calculus is given by

$$\begin{gathered} c1 \equiv c2 \equiv c3 \qquad (i) \\ k = c1^2(c1 - c2 - c3 - c1.c2 - c1.c3 + c2.c3 + c1.c2.c3) \qquad (ii) \\ x \equiv k \qquad (iii) \end{gathered} \tag{9-8}$$

Although this clock calculus is correct, we are not able to prove that $k \neq 0$, so that the clock calculus cannot prove that x is indeed fetched.

This is in fact a good example to introduce the reader to a dynamical extension of the clock calculus. Let us investigate further the table 5 where the image of the generators by the map CLOCK is defined. Assume now we keep unchanged the image of the *boolean delays*. Then, (9-8) is replaced by another calculus we shall write using a single clocked version of the language SIGNAL, where $\perp$ is considered as a value. The program is

$$\begin{gathered} ((c2 := \$c1) \& (c3 := \$c2) \& (c1 := \$c3) \& \\ (k := c1^2(c1 - c2 - c3 - c1.c2 - c1.c3 + c2.c3 + c1.c2.c3)) \& \\ \textbf{synchro}(x, k))\ @c1, c2, k \end{gathered} \tag{9-9}$$

It is easy to check on this example that this process is periodic of period 3 (check the dependence graph: it exhibits a chain of 3 interconnected \$'s), so that *a three step simulation will provide us with a full proof of the program* (9-9). To know whether x will be used or not relies on the inspection of the initial conditions in the memories of the process.

9.4.2 Discussion.

In fact, to refuse to cut the boolean delays in the construction of the dynamical clock calculus of

a process is not a good way, since *it is not true that all the corresponding dynamical clock calculi will be periodic processes with finite period " The simplest example is the program*

```
( ( COUNTMOD ? x:y ! c1:c1', c2:c2', c3:c3' ) &

( COUNTMOD ? x:z ! c1:c1", c2:c2", c3:c3" ) &

( c := c1' default c1" ) )

@ c1', c1"
```

It is easy to check that *the corresponding dynamical clock calculus is not periodic* due to the presence of the generator **default** which causes the boolean c to be aperiodic: no proof can be provided about the values of c. More complex situations can be even provided, where two periodic boolean dynamical systems are interconnected in some input-output mode: the following program is such an example, where the inputs are y and u, of unspecified synchronisation.

```
( ( COUNTMOD ? x:y ! c1:c1', c2:c2', c3:c3' ) &

(k := tt(c1')) & (z := u default k) &

( COUNTMOD ? x:z ! c1:c1", c2:c2", c3:c3" ) )

@ c1', k, z
```

One can check that the second counter is subordinated to the first one in a non periodic fashion.

CONCLUSION: The convenient way to introduce some dynamics in the clock calculus is as follows. Assume a process is given:

1. Built its clock calculus in the classical sense.
2. If the calculus exhibits constraints or relationships that cannot be proved without reasoning about boolean values, try to prove these by inspecting the periodic boolean subprocesses which might be involved in this process.

With such a tool, it might be possible to tune internal counters on external inputs, and to reason about them, exactly as for the time-multiplexer; this turns out to be an opening avenue to *the use of the language SIGNAL itself to describe the machine implementation of SIGNAL programs* , without the need of generating extra clock signals to synchronize the internal clocks of this machine. This will be the subject of future work.

Chapter Ten
CONCLUSION.

The current[10] status of SIGNAL is the following. A version V0 has been developped using the software environment MENTOR [Donzeau-Gouge & al. 1983]. This version provides a subset of the primitives we have studied, together with a set of macros based on the kernel of SIGNAL we have described; instructions are also provided to allow structural programming (for example, the masking of signals is introduced). Regular arrays of processes are also provided, which replace the usual notion of «loop». However, this version does not implement the algebraic clock calculus we have described: the reason is that this calculus was obtained quite recently. A logic based clock calculus was implemented instead, which is not fully satisfactory, since some of the timing problems are solved somewhat arbitrarily. A restricted dependence graph is also implemented, which cannot handle closed paths. Based on this restricted clock calculus, a graph automaton is nevertheless derived using the algorithm we have presented. This version V0 produces executable FORTRAN code. A version V1 is currently developped, which will implement the algebraic clock calculus we have presented.

Our major claim in the introduction was that the executable language SIGNAL is also a system to reason about time and concurrency. Let us further discuss these points. First, several examples have illustrated the power of the clock calculus to investigate and prove timing properties of SIGNAL processes. Second, the conditional dependence graph is a key tool to study concurrency: for example, actions can be identified that never occur simultaneously, a property which can be used for an efficient implementation.

Further developments are the following:

- Using SIGNAL as a high level entry point of a CAD chain from task to machine implementation. The conditional dependence graph is a key tool to study task allocations in a multiprocessor implementation. On the other hand, the ability of SIGNAL to tune faster internal clocks to the environment, allows to simulate the running of internal machine counters which successively fire the actions. In fact, the graph automaton of a SIGNAL process can also be specified in SIGNAL itself. It is our opinion that this feature of SIGNAL, together with new algebras that are currently developped to study the timing in a *quantitative* way should provide a powerful set of tools to computer aided implementation of real-time oriented tasks; see for example [Caspi & Halbwachs 1982, Cohen & al 1985]. Further developments around the dynamical clock calculus we have

---

[10] May 1986

briefly introduced when analysing the case of the time - multiplexer will be required for this purpose.

- Investigating the simplest implementation of the «single token pass» data - flow communication scheme SIGNAL is based upon. Such interfaces would allow to use any type of machine (Von Neumann or Non Von Meumann) as elementary processors inside a multiprocessor architecture with SIGNAL as programming language. We follow here the idea which was at the origin of the *Transputer* which uses OCCAM as a programming language.

- Finding the smallest extension of SIGNAL which would allow to specify any real - time kernel. The corresponding extension would then provide all the capabilities of classical real - time languages, while keeping all the proof properties of the present version available.

- Investigating extensions of SIGNAL including dynamic creation of processes. The interest lies in the use of such extensions in applications which are not properly real - time. For example, continuous speech recognition is certainly «real - time» from the user's viewpoint, but not from the formal one, since the depth of the memory on past data is much too large to allow an easy programming with strict real - time languages such as ESTEREL, SIGNAL, or LUSTRE. It would be desirable to provide such an extension with criteria to recognize if a program is can be actually transformed in a classical SIGNAL program, thus making the proof system of this language still available, while increasing the comfort of the programming.

Acknowledgements: We must first thank the group working on the development and experimentation of SIGNAL, namely Loic Besnard, Patricia Bournai, Thierry Gautier, Bernard Le Goff, and Yves Sorel (at INRIA Rocquencourt), nothing would have been possible without their support. We must thank also the group of Michel Sorine and Yves Sorel at INRIA - Rocquencourt; this group participates in a joint project with the author's group on CAD tools for signal processing and control algorithms implementation, their contribution concerns the development of the whole methodology on significant applications, using the RTL hardware description language ISPS from Carnegie - Mellon university. Finally, we would like also to thank Philippe Darondeau for valuable discussions and helpful suggestions.

REFERENCES.

[Ackermann & Dennis 1979]: W.B. Ackermann, J.B. Dennis, «VAL – A Value Oriented Algorithmic Language, preliminary reference manual», Lab. Comput. Sci., M.I.T. Cambridge, Tech. Rep. TR-218, June 1979.

[ADA 1980]: *Reference Manual for the ADA Programming Language*, CII Honeywell – Bull.

[Benveniste 1985]: A. Benveniste, «A Model to Analyse the Causality in Synchronous Real – Time Systems», INRIA Res. Rep. No 411, Rocquencourt, France.

[Bergerand & al. 1985]: J.L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, E. Pilaud,«Outline of a real – time Data – Flow Language», in *Real Time Systems Symposium*, San Diego Dec.1985.

[Berry & Cosserat 1984]: G. Berry, L. Cosserat, «The ESTEREL Programming Language and its Mathematical Semantics», INRIA Res. Rep. No 327, Rocquencourt, France, to appear in *Science of Computer Programming.*

[Boudol & al. 1984]: G. Boudol, D. Austry, «Algèbre de Processus et Synchronisation», Theoretical Comp. Sc. 30, 91 – 131.

[Brock & Ackerman 1981]: J.D. Brock, W.B. Ackerman, «Scenarios, a model of non determinate computation», Conf. Formal Definition of Programming Concepts, Lect. Notes on Comp. Sc. vol 107, Springer V.

[Brookes & al. 1984]: S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, «A Theory of Communicating Sequential Processes» J. ACM vol 31 no 3, 560 – 599

[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* , 150 – 159.

[Chase 1984]: M. Chase, «A Pipelined Data – Flow Architecture for Digital Signal Processing: the NEC μPD7281 » in *VLSI Signal Processing*, IEEE Press, New York.

[Cohen & al. 1985]: G. Cohen, D. Dubois, J.P. Quadrat, M. Viot, «A Linear – System – Theoretic View of Discrete Event Processes amd its use in Performance Evaluation in Manufacturing», IEEE Trans. on AC, vol AC – 30 .No 3, 210 – 220.

[De Bruin & Boehm 1985]: A. De Bruin, W. Boehm, «The Denotational Semantics of DynaMIC = Dynamic Network of Processes», ACM Trans. on Prog. Lang. and Syst., vol 7 No 4, 656 – 679.

[Dennis 1974]: J.B. Dennis, «First Version of a Data – flow Procedure Language», in *Programming Symp.: Proc. Colloque sur la Programmation*, Paris, France, in *Lect. Notes in Computer Sc.* vol 19, B. Robinet Ed., Springer V., 362 – 376.

[Donzeau – Gouge & al. 1983]: V. Donzeau – Gouge, G. Kahn, B. Lang, B. Mélèse, E. Morcos,

«Outline of a Tool for Document Manipulation», in *Information Processing 83*, R.E.A. Mason Ed., North Holland, 615-620.

[Gaudiot 1985]: J-L Gaudiot, R.W. Vedder, G.K. Tucker, D. Finn, M.L. Campbell, «A distributed VLSI Architecture for Efficient Signal and Data Processing», IEEE Trans on Computers, C-34 No 12, Dec.1985.

[Hoare 1978]: C.A.R. Hoare, «Communicating Sequential Processes», Comm. ACM 21(8), 666-678.

[Kahn 1974]: G. Kahn, «The semantics of a Simple Language for Paralle Programming» in *Proceedings IFIP 74* , J.L. Rosenfeld Ed., North Holland, Amsterdam, 471-475.

[Kahn & Mc Queen 1977]: G. Kahn, D.B. Mac Queen, «Coroutines and Network of Parallel Processes» in *Proceedings IFIP 77*, B. Gilchrist Ed., North Holland, Amsterdam, 993-998.

[Lamport 1985]: L. Lamport, «An Axiomatic Semantics of Concurrent Languages», NATO ASI Series, vol F13, Logic and Models of Concurrent Systems, K.R. Apt Ed.

[Le Guernic & al. 1985]: P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier, «SIGNAL: a Data-Flow oriented Language for Signal Processing», INRIA Res. Rep. No 378, Rocquencourt, France.

[Le Guernic & al. 1986]: P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier, «SIGNAL: a Data-Flow oriented Language for Signal Processing», IEEE Trans. on ASSP, ASSP-34 No 2, 362-374.

[LTR 1978]: *LTR Manuel Officiel de Référence*, Ministère de la Défense, France.

[Milner 1980]: R. Milner, *A Calculus of Communicating Systems*, Lect. Notes in Comp. Sc. vol 92, Springer V.

[OCCAM 1983]: *OCCAM Programming Manual*, INMOS Limited.

[Oppenheim & Schaffer 1975]: A.V. Oppenheim, R. Schaffer, «*Digital Signal Processing*», Englewood Cliffs, NJ, Prentice Hall.

[Plotkin 1981]: G.D. Plotkin, «A Structural Approach to operational Semantics», Lect. Notes, Aarhus Univ.

[Pnueli 1977]: A. Pnueli, «The Temporal Logic of Programs», Proc. of the IEEE Symposium on the Foundations of Computer Science, Providence, Rhode Island.

[Tanzi 1985]: J.M. Tanzi, «Traduction Structurelle des Programmes ESTEREL», Thèse de Troisième cycle, Univ. Nice, Nov. 1985.

PI 287 **A new statistical approach for the automatic segmentation of continuous speech signals**
Régine André-Obrecht – 38 pages ; Mars 86.

PI 288 **Synthèse sur les réseaux locaux temps-réel**
Philippe Belmans – 40 pages ; Mars 86.

PI 289 **Calcul distribué d'un extrémum et du routage associé dans un réseau quelconque**
Jean-Michel Hélary, Aomar Maddi, Michel Raynal – 36 pages ; Mars 86.

PI 290 **A new matrix multiplication systolic array**
Patrice Quinton, Brigitte Joinnault, Pierrick Gachet – 12 pages ; Avril 86.

PI 291 **Une introduction à quelques techniques du contrôle distribué à travers un exemple**
Noël Plouzeau, Michel Raynal, Jean-Pierre Verjus – 22 pages ; Avril 86.

PI 292 **Distributed Synchronization of Parallel Programs : why and how ?**
Patrice Quinton, Jean-Pierre Verjus – 16 pages ; Avril 86.

PI 293 **Sur l'utilisation des séquences multi-images en robotique**
Lionel Marcé – 16 pages ; Avril 86.

PI 294 **Stabilité de l'analyse factorielle des correspondances multiples en cas de données manquantes et de modalités à faibles effectifs**
Habib Bénali – 16 pages ; Avril 86.

PI 295 **The problem of diagnosis with respect to physical parameters for changes in structures**
Georges V. Moustakides – 14 pages ; Avril 86.

PI 296 **An overview of local environment sensing in robotics applications**
Bernard Espiau – 38 pages ; Mai 86.

PI 297 **Context-dependent determiners in logic programming : semantic representations and properties**
Patrick Saint-Dizier – 54 pages ; Mai 86.

PI 298 **Real-Time, Synchronous, Data-Flow Programming : the Language SIGNAL and its Mathematical Semantics**
Paul Le Guernic, Albert Benveniste – 112 pages ; Juin 86.

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