Hybrid dynamical systems theory and the language "SIGNAL"

Albert Benveniste, Bernard Le Goff, Paul Le Guernic

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 838

HYBRID DYNAMICAL SYSTEMS THEORY AND THE LANGUAGE " SIGNAL "

Albert BENVENISTE
Bernard LE GOFF
Paul LE GUERNIC

AVRIL 1988

RR 838

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE N° 403
AVRIL 1988
116 Pages

HYBRID DYNAMICAL SYSTEMS THEORY

AND THE LANGUAGE "SIGNAL"

Albert BENVENISTE, Bernard LE GOFF, Paul LE GUERNIC
IRISA/INRIA
Campus de Beaulieu
35042 RENNES CEDEX, FRANCE

Summary: *we study the logic and synchronization characteristics of general dynamical systems called Hybrid Dynamical Systems. Our theory generalizes the notion of Discrete Event Dynamical Systems by handling numerics as well as symbolics. Our theory is supported by the programming language SIGNAL and a mathematical model of relational style. This framework allows us to formulate in the same way HDS programming or specification and HDS control. The core of the theory is the notion of HDS resolution which is based on a reduction technique mapping any HDS specification program into a polynomial dynamical system on the finite field of integers modulo 3; all the algorithms are then based on the study of this dynamical system. This report is divided into two parts; the first one is devoted to the discussion of the motivations, the presentation of two examples, and an informal presentation of the theory; in the second part, the mathematical theory (models and algorithms) is presented.*

THEORIE DES SYSTEMES DYNAMIQUES HYBRIDES:

LE LANGAGE "SIGNAL".

Résumé: *Dans ce rapport, nous étudions les systèmes dynamiques généraux du point de vue de leurs propriétés de synchronisation et de logique; sous ce point de vue, les systèmes dynamiques généraux seront appelés «systèmes hybrides». Notre théorie généralise la notion de Système à Evènements Discrets en ce qu'elle manipule simultanément les aspects numériques et symboliques. Elle s'appuie sur le langage synchrone SIGNAL, et est fondée sur un modèle de type relationnel. Ce point de vue nous permet d'embrasser tout à la fois programmation, spécification, et contrôle des systèmes dynamiques hybrides. Le coeur de la théorie est la «résolution» des systèmes hybrides, résolution s'effectuant par le biais d'une technique de réduction à un calcul de système dynamique polynomial sur le corps fini des entiers modulo 3. Ce rapport se compose de deux articles jumeaux; le premier est consacré à une présentation informelle assortie de la discussion d'exemples, tandis que le second présente les aspects proprement mathématiques de la théorie.*[1]

---

[1] Ce travail se place au sein de l'accord-cadre CNET-INRIA *poste de travail pour le traitement du signal et la conduite de processus*

Hybrid Dynamical Systems theory and the language SIGNAL.

Part I: motivation, examples, and informal presentation.

Albert Benveniste, Paul Le Guernic
IRISA/INRIA, Campus de Rennes Beaulieu
35042 RENNES CEDEX, FRANCE

Summary: *we study the logic and synchronization characteristics of general dynamical systems called Hybrid Dynamical Systems. Our theory generalizes the notion of Discrete Event Dynamical Systems by handling numerics as well as symbolics. Our theory is supported by the programming language SIGNAL and a mathematical model of relational style. This framework allows us to formulate in the same way HDS programming or specification and HDS control. The core of the theory is the notion of HDS resolution which is based on a reduction technique mapping any HDS specification program into a polynomial dynamical system on the finite field of integers modulo 3; all the algorithms are then based on the study of this dynamical system. This first part is devoted to an informal discussion.*[1]

---

[1] this work is supported by the *accord-cadre CNET-INRIA "Poste de Travail pour le traitement du signal et la conduite de processus"*

Table of Contents

Chapter One
Introduction.

1.1 Requirements from applications: Hybrid Dynamical Systems (HDS).

As recognized in [Levis et al. 1987] and [Ho 1987], most modern applications involving dynamical systems are very complex in nature; think of

- real−time complex control or signal processing systems in avionics, aeronautics, and in C^3−military systems,
- automation handling man−machine interfaces of control systems (monitoring, trouble shooting, visual aids in avionics, remote manipulation...)
- vision− and sensory−based control in robotics,
- complex pattern recognition applications such as continuous speech recognition

to mention just a few. Some particular features of these applications are

1. the mixed continuous/discontinuous nature of time because of the simultaenous presence of familiar differential/difference dynamical subsystems and discrete event systems relating these subsystems;
2. the presence of dynamics;
3. a large combinatorial complexity as far as logic and synchronization is concerned (there is no concise model to describe precisely such applications), hence the need for modularity.

On the other hand, the highly demanding nature of these applications forces to consider as well the requirement of highly efficient and reliable implementation, a goal which can be achieved only by using parallel or distributed implementations. But the modular structure of the application is generally different from the modular structure of the implementation, hence the need for a theory to transform such dynamical systems into equivalent ones with a different structure.

Discrete Event Dynamical Systems (DEDS) have been introduced as a theoretical framework for the study of flexible manufacturing and related systems by Wonham and Ramadge [Ramadge and Wonham 1987, a−b], and have been widely studied since their introduction. Roughly speaking, DEDS are finite state transition systems which are observed and can be controlled by the language generated by the labels that are attached to each transition, regardless of the precise meaning of these labels. However, in most of the above mentioned applications, some

actions involving possibly complex numerics can influence transitions (e.g. when some internally generated signal exceeds a given threshold). This restriction makes the use of the DEDS approach not suitable to the global study of complex dynamical systems of mixed nature such as listed above.

On the other hand, the area of computer science has developped a very large and deep activity to handle such complex dynamical systems in the areas of *real-time systems and languages*, [Young 1982], or *communicating systems* with CSP [Brookes & al. 1984] and CCS [Milner 1980] [Milner 1983] as most famous examples. More recently, the new approach of ***synchronous programming*** has been introduced and developped around the languages ESTEREL ([Berry & Cosserat 1984] [Gonthier 1988]), LUSTRE ([Bergerand & al. 1985]) and SIGNAL ([Le Guernic & al. 1986] [Le Guernic & Benveniste 1986] [Benveniste & Le Guernic 1987]). The techniques of synchronous programming cover the kind of dynamical systems we have listed above. This is the direction we want to pursue and further discuss from a control viewpoint in this article.

In the sequel, ***Hybrid Dynamical Systems (HDS)*** theory will refer to a theory handling synchronization, logic, and their interconnections to numerics in dynamical systems. As the reader will understand while reading this paper, the mixed nature of HDS make them definitely more difficult to study than DEDS, which justifies the development of a new theory and paradigm.

1.2 A new paradigm.

1.2.1 Building complex objects requires the use of languages.

Our first remark was about the highly combinatorial complexity of HDS. Such a complexity faces us with a new problem which was not considered before in the control community, namely the difficulty of simply *describing* or *constructing* HDS. This point is clearly illustrated by the use of sophisticated operating system facilities in continuous speech recognition systems, or the interest for the use of «expert systems» as a clean tool to describe the «spaghetti» structure of the bench of rules required to specify and program an intelligent PID controller with its set of heuristics [Astrom & al. 1986]. On the other hand, it should be clear that the direct use of the formalisms a la Ramadge and Wonham is not the easiest way to describe directly complex applications. In fact, this has been for a long time recognized by computer scientists as a sufficient reason for introducing programming languages, i.e. *concrete syntax* with the usual hierarchical constructs. The core of our approach is the kernel of the language SIGNAL, which is composed of only 5 instructions to describe any HDS.

1.2.2 Why relational languages?

A second claim is that a HDS should be described via a set of *relations* or *constraints*, rather than as a complicated input-output map as usually in control science. In fact, the following

argument will be present permanently in this paper, and has been already recognized by J.C. Willems [Willems 1987] for the theory of linear dynamical systems:

modular construction of HDS
⇓
interconnecting given HDS's yields an implicitely defined HDS
⇓
HDS should be defined via constraints on the set of all possible behaviours

Hence our approach as well as the language SIGNAL will have a relational flavour.

1.2.3 The basic problem: HDS resolution.

An immediate consequence of this choice is that such HDS specifications cannot be effective, i.e. it is not immediately possible to compute the outputs of a so-specified HDS in response to some sequences of inputs. The control scientist will recognize a standard situation when handling descriptor or implicit linear dynamical systems. By ***HDS resolution***, we have in mind a procedure to transform any relational HDS specification into a machine which can execute the desired behaviours, and thus represents the desired equivalent input-output map. Hence, HDS resolution has some flavour of realization theory, where the program describing a given HDS is the external representation, while the associated executable machine is the corresponding state machine internal representation. Here appears one of the most fundamental bridges between control and computer science in HDS theory: building a compiler for the language SIGNAL is equivalent to looking for a minimal realization of the so-specified HDS.

1.2.4 What is the nature of control problems for HDS?

Another consequence of this point of view is that *exact model following control problems* (such as considered by Ramadge and Wonham) are just particular cases of HDS resolution, since requiring an exact model following is just achieved by adding further constraints described within the same framework as the HDS itself. This point will be discussed in the examples below.

1.2.5 What is the nature of time for HDS?

Complex applications such as mentioned above are inherently distributed in nature. Hence every subsystem possesses his own time reference, namely the ordered collection of all the communications or actions this subsystem performs: in sensory based control systems, each sensor possesses its own digital processing with proper sampling rate, actuators generally have a slower sampling rate than sensors, and moreover the software devoted to monitoring only reacts to various kinds of alarms that are caused by internally or externally generated indicators. Hence the nature of time in HDS is by no means universal, but rather local to each subsystem, and consequently multiform. A fundamental consequence is that communications between

subsystems impose constraints on the timing of these subsystems: an alarm can be sent by an actuator or a sensor to the supervisor which turns out to react on actuators: the whole result is a constraint between the clocks of these subsystems. Hence handling these multiform time references and reasoning about them is one of the fundamental tasks we have to perform. In the sequel, signals possessing identical time references will be said to have the same ***clock***.

1.3 Some features of our approach, and organization of the paper.

To achieve our goals, we need both a programming language (SIGNAL) and a mathematical model to develop the algorithms for HDS resolution. Here is the kind of scheme we shall follow:

algebra of SIGNAL programs
↓ (*modelling*)
mathematical model of HDS algebra
↓ (*projection*)
HDS subalgebra
↓ (*resolution*)
executable machine algebra

where every ↓ denotes some homomorphism between convenient algebras we shall introduce later. The actual algorithm for SIGNAL program compilation will directly consider the composition of the first two maps, thus yielding the scheme

algebra of SIGNAL programs
↓ (*« coding »*)
HDS subalgebra
↓ (*resolution*)
executable machine algebra

Chapter Two
The programming language SIGNAL–kernel; two examples.

As we have indicated before, HDS should be specified via programming languages. We shall now present such a language: a subset of the language SIGNAL[1]. To be concise, we shall introduce only the primitives of the language, and drop any reference to typing and various declarations; the interested reader is referred to [Gautier 1987].

2.1 SIGNAL-kernel as a specification language for HDS.

SIGNAL handles (possibly infinite) sequences of data with time implicit: such sequences will be referred to as ***signals***. For example, x denotes the infinite sequence $\{x_t\}_{t\geq 1}$ where the time index t is attached to this signal; signals possessing the same time index are said to have the same ***clock***, so that clocks are equivalence classes of simultaneous signals (a formal definition is discussed in the companion paper [Benveniste & al. ** 1988]). Instructions of the language SIGNAL are intended to relate clocks as well as values of the various signals involved in a given HDS. We shall often use the name of ***program*** or ***process*** to denote a system of such relations; processes will be used as black-boxes. All these notions are introduced formally in the companion paper [Benveniste & al. ** 1988].

A basic principle in SIGNAL is that a single name is assigned to every signal, so that in the sequel (and unless explicitely stated), identical name refers to identical signals. The kernel-language SIGNAL possesses 5 instructions, the first of them being a generic one.

p(x(1),...,x(n))	(i)
y := x$ ***init*** x_0	(ii)
y := x ***when*** b	(iii)
y := u ***default*** v	(iv)
P | Q	(v)

Their intuitive meaning is the following:

(i): direct extension of instantaneous relations into relations acting on flows:

$$p(x(1),\ldots,x(n)) \Leftrightarrow \forall t: p(x_t(1),\ldots,x_t(n))$$

Examples are functions such as z:=x+y ($\forall t: z_t = x_t + y_t$) or statements such as (a and b) or c := *true* ($\forall t: (a_t \text{ and } b_t) \text{ or } c_t = true$). A byproduct of this instruction is that *all referred signals must have the same time index, i.e. must be present simultaneously.* This is a generic instruction, i.e. we assume a family of relations is at hand. If one chooses an instantaneous relation

---

[1] SIGNAL is a joint trademark of CNET and INRIA

accepting any n-uple, the resulting SIGNAL instruction only constrains the involved signals to have the same clock: the so-obtained instruction written ***synchro*** x,y,... has thus as only effect to force the two signals x,y,... to have the same clock.

(ii): logical delay (the usual operator z^{-1})

y := x$ *init* $x_0 \Leftrightarrow \forall t > 1 : y_t = x_{t-1}, y_1 = x_0$

(iii): condition (b is boolean): y equals x when the signal x and the boolean b are available and b is true; otherwise, y is not emitted; the result is an event-based undersampling of signals.

(iv): y merges u and v, with priority to u when both signals are simultaneously present; this instruction is the key to oversampling as we shall see later.

The instructions (i-iv) specify the elementary processes, we shall call *generators*. The objects named x, y, u, v, b, will be called *signals*.

(v): communication of already defined processes: P and Q communicate through their signals with common names; for example

y := zy + a | zy := y$ *init* x_0

denotes the system of recurrent equations for $t \geqq 1$

$$y_t = zy_t + a_t$$
$$zy_t = y_{t-1}, zy_1 = x_0$$

which is equivalent to $y_t = y_{t-1} + a_t, y_0 = x_0$.

2.2 The shared track example.

This example is borrowed from [Ostroff 1986].

2.2.1 Informal description.

Consider a plant which consists of two diesel trains which share a common section of railway track. On the shared track, there is a diesel pump for refueling the trains. The train fuel can hold up to 1000 gallons of diesel measured in whole gallons. There is an automatic mechanism which allows a controller to sample the level of diesel in the tank of the train, and a facility exists for the controller to command the pump to deliver any amount of fuel to the train. Since we wish to prevent the disastrous situation of two trains simultaneously occupying the shared track, two traffic lights have been installed. Each train is allocated a traffic light at its entrance

to the shared track until he receives the signal to move, at which it immediately proceeds onto the shared track. We shall now discuss the SIGNAL programming of this example.

2.2.2 Some basic mechanisms.

To allow an easy description of complex objects, we want to build a toolbox of elementary mechanisms which will be used similarly to basic instructions in the sequel (something like «macros»). In this paragraph, to avoid the need for explicit typing, we shall use the following generic notations

- u,x,y,z...: signal of any type
- a,b,c...: boolean signals
- h,k,l: signals of type *event*, i.e boolean signals which take only the value *true*

Here follows a first example.

(1) Access to the clock of a signal.

h := ***event*** x
=
h := (x=x)

The pure clock h is delivered when x is present (since x=x always holds).

In this example, the notation

processname
=
body of the process

is used in order to give a name to a process defined by a set of formulas. Here, «processname» is of the form

list of output signals := NAME (*list of input signals*)

Using this notation, we present further basic mechanisms.

(2) Extraction of the occurrences *true* of a boolean signal

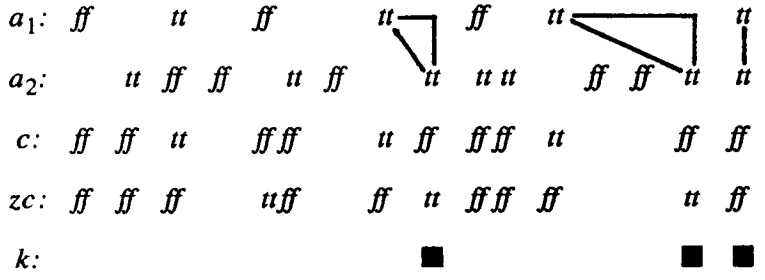
h := ***when***(b)
=
h := b ***when*** b

(3) A synchronized memory

y := x *cell* b *init* y_0
=
zy := y$ *init* y_0
y := x *default* zy
synchro y, (x *default when*(b))

The output y returns either the present value of x (when x is received), or the last received value of x when b is present and true.

(4) A guard on interleaved signals («followed by»). The event k is emitted when $a_2 = true$ occurs simultaneously with $a_1 = true$ or immediately after $a_1 = true$ when the latter is present alone. Here follows a picture of the corresponding temporal behaviour; the internal signals of the program are also depicted:



The diagram above shows how the various signals are interleaved; symbols appearing in the same column are delivered simultaneously. An intuitive reading of this picture is that ■ should occur exactly when triangles linking occurrences of *tt* can be inserted without intersecting other symbol. Notice that the space between the columns have no interpretation in terms of some «regular» physical time, this diagram only specifies the global interleaving of the various signal flows. Here follows the program:

k := a_1 *fby* a_2
=
k := *when*(a_2) *when* (a_1 *default* zc)
zc := c$ *init false*
c := (*not when*(a_2)) *default* a_1

An interesting use of this instruction is the follwing relation defined between the boolean signal a and the clock k:

k := a *fby* k

which means that the event k can (but is not obliged to) occur once simultaneously or

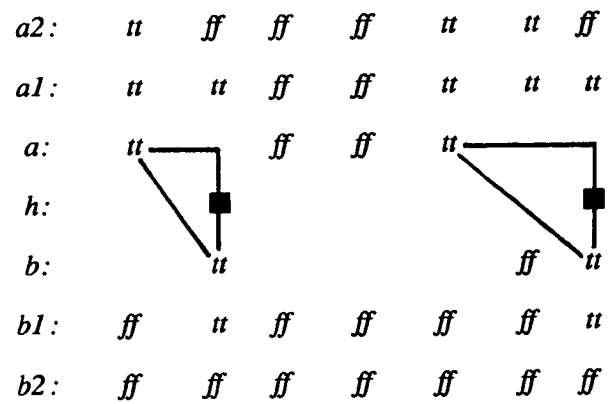
immediately after every occurrence of a=*true*. Here follows a picture of two different and both admissible behaviours of this program:

and

(5) An unordered guard.

```
h := occurred(a,b)
        =
     a1 := a default (a2$ init false)
     a2 := (not h) default a1
     b1 := b default (b2$ init false)
     b2 := (not h) default b1
     h := when(a1) when b1
     synchro a1, b1, a default b
```

The event h occurs when an occurrence of a=*true* is eventually followed by an occurrence of b=*true* (here, «eventually» means that a=*false* does not occur between the two events mentioned above), or vice–versa; but only a single use of any of these occurrences of a=*true* or b=*true* is allowed to produce h. Here follows a picture of this program where desired events are again indicated by triangles:



2.2.3 Defining subprocesses, or black-boxes.

The preceding technique is convenient to relate a few different signals via a single new macro; this is what control scientists usually perform by plugging a few equations into a single formula. However, when the system's complexity increases, it is usually preferred to use block-diagram descriptions; the notion of SIGNAL process is the convenient construct for block-diagram building. Here follows the corresponding notation

```
NEW_PROCESS (list of parameters) { list of visible signals}
          =
     OLD_PROCESS_1 list of a_1: b_1
|    ....
|    OLD_PROCESS_n list of a_n: b_n
end
```

where a:b means that the signal a of OLD_PROCESS is renamed b; this mechanism is the basis for black-box interconnection in block-diagram building since signals denoted by identical names must be identical. Signals which appear in the body of the process but are not listed in its interface are local signals, i.e. are invisible from outside. To facilitate the reading, it will be useful to mark some of the interfaces with ? or ! to mention that the corresponding signal interfaces are *input* or *output*; this will be used in the sequel.

2.2.3.1 A decreasing counter with reset.

```
DECCOUNT { ? integer reset ! bool empty}
          =
     n := reset default (zn - 1)
|    zn := n$ init 0
|    empty := when(n = 0) default (not event(reset))
|    synchro (when(zn = 0)), (reset when zn = 0)
```

When the integer signal "reset" (which is assumed to be >0) is received, the counter restarts decreasing from this value, until the counter reaches 0. The boolean «empty» delivers *false* when the counter is reset, and *true* when the counter gets empty. The last instruction ensures that, when zn=0 occurs, then «reset» must be present, i.e. the counter does not decrease below zero. The first occurrence of the signal n is when «reset» is received. Finally, notice that nothing prevents the counter to be reset before being empty. This program will be used repeatedly to represent the consumption of fuel in the tank of the train, or the delivery of fuel by the pump on the shared track.

2.2.3.2 A train on travelling.

```
TRAIN (int Max_level, min_level) {? event start ! bool ask_fuel }
          =
```

```
    reset := (Max_level - min_level) when start
|   DECCOUNT empty: ask_fuel
```

The train travel is simply modelled by the consumption of fuel until the tank is empty. «ask_fuel» delivers *false* when refueling starts, and *true* when the tank is empty. Here, we used renaming of the output of the process DECCOUNT. Note that "Max_level" and "min_level" are *constants*, i.e. are permanently available.

2.2.3.3 The shared track.

```
SHARED_TRACK (int Max_level, min_level) {? event train1_on, train2_on ! bool empty}
        =
    tank1 := (Max_level - min_level) when train1_on
|   tank2 := (Max_level - min_level) when train2_on
|   reset := tank1 default tank2
|   DECCOUNT
```

Here, the program DECCOUNT represents the shared pump; the fact that the pump is shared is taken into account by the use of the merge operator ***default*** to produce the signal "reset" for the pump to start; this operator merges the demands of the two trains. The rest of the program should be selfevident. Here, the internal signal "reset" is different from the "reset" signal occurring in the TRAIN subprocess: they take the same value, but are delivered at different instants. This is of no importance, since, by convention, internal signals (i.e. signals which are not listed in the name of the process) are not visible from outside.

This program specifies the shared track mechanism. No lights are provided for the moment to prevent from possible collisions.

2.2.4 The whole program.

Here, renaming is used again and again. We introduce also the technique used in SIGNAL to allow modular programming (i.e. to build block–diagrams from predefined black–boxes). Here follows the program SNCF[2]. For the sake of clarity, we have omitted the declarations of parameters.

---

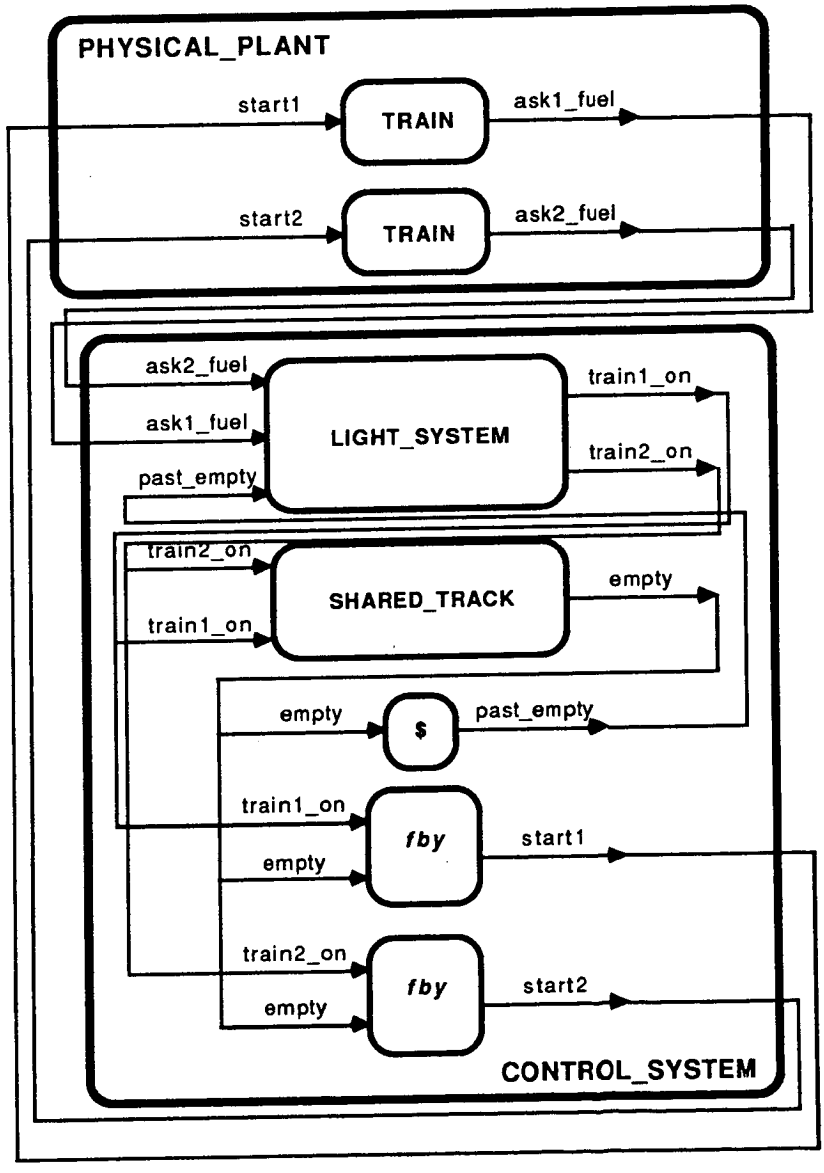
2 Societe Nationale des Chemins de Fer Francais, a trademark from French government

```
SNCF {? %no input% ! %what you want to observe% }
        =
     PHYSICAL__PLANT
|    CONTROL__SYSTEM

where

     PHYSICAL__PLANT { ? event start1, start2 ! bool ask1__fuel, ask2__fuel }
             =
          TRAIN start: start1, ask__fuel: ask1__fuel
|  .       TRAIN start: start2, ask__fuel: ask2__fuel

     where

          TRAIN {...} %already seen%

     end

     CONTROL__SYSTEM { ? bool ask1__fuel, ask2__fuel ! event start1, start2 }
             =
          LIGHT__SYSTEM
|         SHARED__TRACK
|         past__empty := empty$ init true
|         start1 := train1__on fby empty
|         start2 := train2__on fby empty)

          where

          SHARED__TRACK {...} %already seen%

          LIGHT__SYSTEM{ ? ask1__fuel, ask2__fuel, past__empty ! train1__on, train2__on }
                  =
               train1__on := occurred(ask1__fuel, past__empty)
|              train2__on := when{ (false when train1__on)
                           default occurred(ask2__fuel, past__empty)}

          end

     end

end
```

The label __i (i=1,2) refers to the particular train. The light system enforces priority to train1; the signal «past__empty» indicates whether the track is free (boolean type). Finally, the last

block of instructions keep track of which train did enter the shared track, hence indicates which train exits the track.

This writing of the program clearely indicates the decomposition into «physical plant» and «control system» as two mutually interacting subsystems. Notice that the program SNCF does not possess any input: it merely specifies relations between its various signals, and is in fact a program *to simulate the pair* { control system, physical plant }. This is a typical example of modular programming of HDS. Here follows a block-diagram corresponding to this program; such a picture shows the kind of graphical interface which is currently developped for the language SIGNAL.



COMMENTS: A proof of the fact that the control system ensures the absence of collision would require the investigation of the whole program SNCF. This is expected to be a typical situation,

and a very unpleasant one for complex systems such as encountered in practice. The fact that SIGNAL is indeed a relational language should be used to allow a new as well as control oriented style of programming.

2.2.5 Another style of programming.

2.2.5.1 A proved program.

Keep the preceding program, except that the subprogram SHARED_TRACK is modified as follows. Instead of using DECCOUNT, we make use of the following program we shall call GUARDED_COUNT:

```
GUARDED_COUNT { ? integer reset ! bool empty}
              =
    n = reset default (zn - 1)
  | zn = n$ init 0
  | empty = when(n = 0) default (not event(reset))
  | synchro (when(zn = 0)), reset

end
```

Notice that the two programs DECCOUNT and GUARDED_COUNT differ only via the last *synchro* instruction. Here, the main difference is that the program GUARDED_COUNT is *active*, i.e. that it refuses any reset signal before being empty: by the way no train can proceed on the shared track while occupied. Hence this program is itself *a proof of the fulfilment of the specification* we wanted, namely a guaranteed collision avoidance. In such a case, the proof of correctness of this program is obtained by verifying that *no deadlock occurs, due to incompatible constraints*. This will be the purpose of the algorithm we shall outline later and fully describe in [Benveniste & al. ** 1988].

2.2.5.2 Towards program synthesis, and pursuing the control viewpoint.

As a matter of fact, it would be even possible to replace the explicit synchronisation mechanism LIGHT_SYSTEM by the implicit synchronisation mechanism we describe now. Keep the whole program SNCF as just before, except LIGHT_SYSTEM is replaced by the following set of instructions where the index i refers to the particular train:

```
train_i_on := ask_i_fuel fby train_i_on
```

This instruction means that train i waits some unknown amount of time after asking for refueling, before eventually proceeding on the shared track. This is exactly the degree of freedom which is required for the strong synchronisation caused by the program GUARDED_TRACK to be accepted. But it turns out that how long the train i waits is not

specified here: an «optimal policy» should be then calculated by the compiler, acting here as a control synthesis system.

2.3 A signal processing example: detecting changes in the mean of a Gaussian signal.

This is a simpler version of a program which has already been written to segment speech signal into *voiced/unvoiced/silent* parts. However the most interesting features of the speech example will be still encountered here.

2.3.1 The Page algorithm.

Consider a signal

$$x_n = \mu_*(n) + v_n$$

where v_n is a standard white noise, and the mean $\mu_*(n)$ is piecewise constant. The objective is to detect the jumps of this mean. A standard algorithm, based on the Page stopping rule [Basseville and Benveniste 1986] is the following one:

1/ INITIALISATION AND PERMANENT LEARNING OF μ:

$$0 < n: \ \mu_n = \mu_{n-1} + \frac{1}{n}(x_n - \mu_{n-1}), \quad \mu_0 = 0 \tag{2-1}$$

This is nothing but the recursive version of the empirical mean of the random variable x_n.

2/ PERFORMING THE TEST (we need to know sufficiently accurately the mean before running the test, hence the latency):

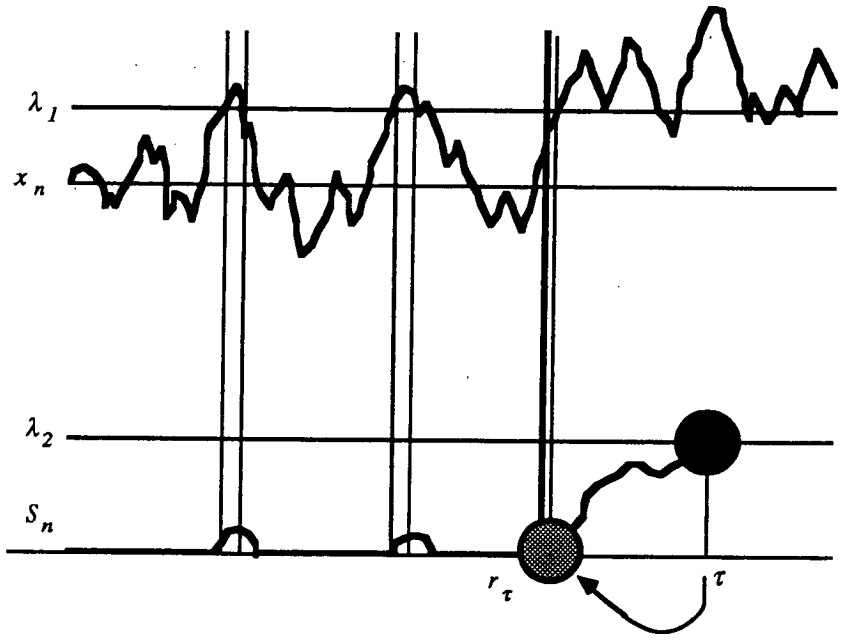
$$\begin{gathered} n > N: \ S_n = (S_{n-1} + x_n - \mu_n - \lambda_1)_+, \ S_N = 0 \\ r_n = \max\{k \leqq n : S_k = 0\} \\ \text{test}: \ S_n \geqq \lambda_2 ? \end{gathered} \tag{2-2}$$

Here, $x_+ = \max(x, 0)$. The λ_i's are positive thresholds; the present algorithm is designed to detect increases of the mean, straightforward modifications are needed for the case of decreasing mean.

3/ RESTARTING (when a detection occurred):

τ = detection instant
restart STEP 1 from instant r_τ

Notice that this is not a formal description of the whole algorithm, since the step 3/ has been formulated informally. This is in fact a very simple but typical example of the difficulty in formally specifying HDS; in fact it should be clear from the two examples we discuss in this paper that the use of languages cannot be avoided. The whole procedure is depicted in the following scheme:



This presentation of the algorithm reveals the following facts about the algorithm:

- it involves 2 tasks
  1. permanent learning of the empirical mean
  2. performing the test
- it proceeds along 3 steps
  1. initialisation: learning of the mean only
  2. learning the mean and performing the test
  3. processing backward the signal up to the estimated change time, where the whole procedure is restarted.

This is a typical FORTRAN (or any of your favorite sequential programming language) point of view on the algorithm. However, due to the backward processing, this is not really suited to real–time execution (a buffer of possibly large size would be needed).

2.3.2 A real–time oriented style of programming.

We can use the fact that the candidate for a restarting of the procedure, namely r_n, is permanently available. Hence, when $r_n < n$ holds (notice that r_n keeps constant in this case as long as $S_n>0$ holds), we can fire an auxiliary copy of the learning procedure for the mean μ. As a consequence, when a detection occurs, the auxiliary procedure is ready to become the main one: this is achieved thanks to a switcher mechanism. This new way to consider the algorithm does require no buffer any more, and hence is suitable to real–time implementation. Unfortunately, a FORTRAN programming of this kind of mechanism is quite unpleasant; but we shall see that this is the easiest way to program the algorithm in SIGNAL, thus exhibiting an interesting feature of SIGNAL programming.

Let us present the program.

```
PAGE (integer N; real λ1,λ2) {? real x ! event change}
        =
     LEARNING(N) reset: reset1, mu: mu1, time: time1
|    LEARNING(N) reset: reset2, mu: mu2, time: time2
|    SWITCH
|    TEST(λ1,λ2)
```

where

```
     LEARNING (integer N) {? real x; event reset ! integer time; real mu}
              =
          m := zm + (1/t)(x – zm)
|         zm := (0 when reset) default m$ init 0
|         t := zt + 1
|         zt := (0 when reset) default t$ init 1
|         mu := m when t>N
|         time := t when t>N
```

end

%this is the learning of the empirical mean μ according to step 1 by taking into account the reset and the fact that the outputs «mu» and «time» have to be delivered only after the initialisation.%

```
     SWITCH {? event change, reset; integer time1, time2; real mu1,mu2
                          ! event reset1, reset2; integer time; real mu}
              =
          bool := not (bool$ init false)
```

```
    synchro bool, change
    is_main := bool cell reset init true
    reset1 := (change when bool) default (reset when not is_main)
    reset2 := (change when not bool) default (reset when is_main)
    mu := (mu1 when is_main) default (mu2 when not is_main)
    time := (time1 when is_main) default (time2 when not is_main)
end
```

%«bool» is a boolean signal which plays the role of a switcher fired by the detection of changes (the first 2 instructions); «is_main» indicates which copy of the LEARNING program is the main one, this boolean signal is available when the test runs, and keeps track of the values of «bool»%

```
TEST (real λ1,λ2) {? integer time; real x,mu ! event change, reset}
                =
    y := x when event(mu)
    S := max(zS + y - mu - λ1,0)
    zS := ((0 when change) default S)$ init 0
    reset := (time when S=0) cell event mu
    change := when(S>λ2)
end
```

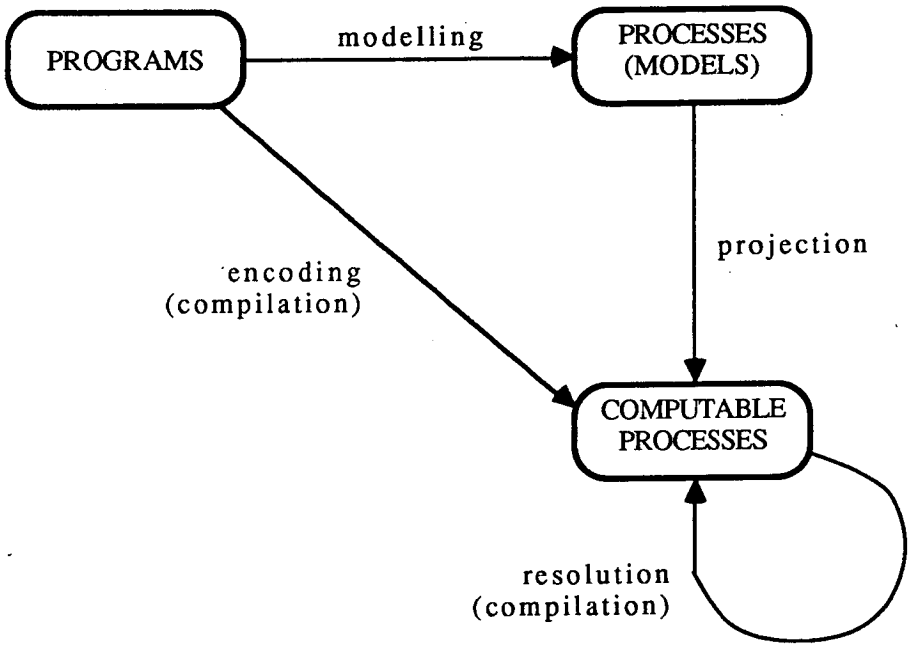
end

%The first instruction synchronizes the input signal «x» with the empirical mean «mu» which is produced only after the initialisation period; otherwise, the program would be wrong (such an error could be detected by the compiler, as we shall see later).%

Again would a picture of the block-diagram corresponding to the main program facilitate the reading; but it should be clear on the other hand that a purely graphical specification is not desired since systems of equations are more convenient at the low level.

2.4 Introducing the principles of HDS modelling.

In the companion paper [Benveniste & al. ** 1988] the problem of HDS modelling is addressed. The method we use is depicted in the following scheme



Let us explain the scheme. Three algebras are mentioned, namely

1. the set of SIGNAL programs, as an algebra of *syntactic objects* (i.e. texts)
2. the algebra of *processes*, which has to be defined as a mathematical object
3. the algebra of *computable* processes, i.e. of the processes for which a *resolution* does exist in the previously mentioned sense.

Considering mathematical models is classical in control science, so that algebras 2. and 3. are not surprising; but we insist again on the fact that, since text is used to specify HDS, a formal mapping from text into mathematical models is needed. Hence the arrows denote homomorphisms between the three algebras, and their intuitive interpretation is indicated on the scheme. In [Benveniste & al. ** 1988], we introduce two different kinds of modelling, i.e. two different algebras of processes to be used for the coding of SIGNAL programs. To introduce the reader to these two style of modelling, here follows a quick review of the computer science viewpoint on the modelling of HDS under the name of «communicating or concurrent processes». There are mainly two classes of styles for models of concurrent processes: the *operational* models and the *denotational* ones.

In the operational style advocated in [Plotkin 1981], processes are dynamic objects which evolve under the action of events in a way which is specified by rules. The operational style could be considered as a hierarchical method to build (possibly infinite) state machines. The operational style makes the study of communication and concurrency of dynamic processes quite easy, and

leads very quickly to effective implementations. To the operational family belong for example SCCS of [Milner 1982], the operational semantics of the synchronous languages ESTEREL [Berry & Cosserat 1984] and SIGNAL [Le Guernic & Benveniste 1986], and also to some extend the failure based CSP model of [Brookes & al. 1984], and the work of Cardelli on analog processes and real time agents ([Cardelli 1980, 1982]). We shall first outline in [Benveniste & al. ** 1988] an approach to modelling of HDS via transition systems and operational semantics.

The denotational approach concentrates on the modelling of input-output maps, following a point of view closer in spirit to the system theoretic framework of the control community. To our knowledge, the pioneering work relevant to denotational style is the Dynamic Network Processes model introduced in [Kahn 1974]. To this family belong also the deep study [De Bruin & Boehm 1985] on Kahn's approach. The advantage of this approach is its elegance, but a severe drawback is the necessity to rely on a difficult *continuation* technique ([De Bruin & Boehm 1985]) to study the communication of processes. To some extend, the approach of [Inan & Varaiya 1987] possesses features of both the above mentioned CSP model as well as the denotational style. Our second approach, also presented in [Benveniste & al. ** 1988], belongs also to this family, although it follows a relational rather than functional point of view.

The next chapter will be now devoted to an informal presentation of HDS resolution. In fact we shall squeeze the modelling step in the diagram above and introduce directly the «encoding» relying on a heuristic justification. Again is the rigorous justification left to [Benveniste & al. ** 1988].

Chapter Three
HDS resolution: an informal presentation.

Here follows the intuitive description of our method. Recall that SIGNAL is obtained by extending a given «instantaneous language» (the set of data types and corresponding relations known by the instruction (i)) with a small set of primitives to obtain a model of dynamical systems. Denoting by IL (Instantaneous Language) this starting point, any homomorphism ψ from IL into itself is easily extended to an homomorphism Ψ from SIGNAL into itself.

In our case, the homomorphism ψ is constructed as follows

STEP 1: among the relations of IL, select the subfamily of relations and corresponding data types for which you accept to solve systems of equations (and are supposed to can!).

STEP 2: other instantaneous relations must be *functions*, and are encoded into their dependence graph; hence corresponding data types are handled as sets of labels for which dependence graphs summarize all the possible rewritings or substitutions.

This defines our homomorphism; its ability to reason about dynamical systems as well as its complexity relies on the choice we have done at STEP 1. Here we shall select the boolean variables together with the boolean relations generated by {:=, ***and, or, not***} and the constants *true, false*; this choice is motivated by the particular role played by the booleans in the instruction ***when***.

3.1 The fundamental homorphism.

3.1.1 Encoding data dependencies for non-boolean objects of IL.

STEP 1: mapping data types.

We introduce the set Λ of ***labels*** (or names); the only relation defined on Λ is the partial orders induced by directed graphs; we shall write $a \longrightarrow b$ to denote that a precedes b; in order to define partial orders, all so-obtained graphs should be acyclic. Then, we introduce the map ψ mapping a variable of any type into its name. The image of a variable a of any type by ψ will be denoted by $\psi(a)$ or, when no confusion can result, simply by a.

STEP 2: mapping non-boolean instantaneous functions into partial orders.

Functions induce partial orders between their labels as follows

$$y = f(x_1, \ldots, x_n) \;\Rightarrow\; \{\psi(x_1) \longrightarrow \psi(y), \ldots, \psi(x_n) \longrightarrow \psi(y)\}$$

simply denoted by

$$y = f(x_1, \ldots, x_n) \Rightarrow \{x_1 \longrightarrow y, \ldots, x_n \longrightarrow y\}$$

The so obtained partial orders summarize all the possible rewritings such as substituting the label y by $f(x_1, \ldots, x_n)$ in other expressions. This coding will be used later. Our next task is to extend this mapping to SIGNAL programs, i.e. to extend this static map into a map on dynamical systems.

3.1.2 Encoding SIGNAL programs.

The image of SIGNAL programs we shall obtain will be referred to as ***synchro-processes***. Synchro-processes are defined via constraints involving clocks, booleans, and labels. We shall provide an algebra with a convenient calculus where the pairs {clocks, booleans} can be represented. All we need to encode are the following status: *absent, present, true, false.* These are encoded onto the finite field $F_3 = Z/3Z$ of integers modulo 3 as follows

$$\begin{aligned} &\textit{true}: +1 \\ &\textit{false}: -1 \\ &\textit{absent}: 0 \\ &\textit{present}: \pm 1 \end{aligned}$$

where ± 1 denotes a non determinate choice of $+1$ or -1; i.e. we handle in the same way clocks and boolean of nondeterminate value. Let us apply this idea to encode SIGNAL programs. The following notation will be used to present this coding:

$$\textit{synch}\,(\text{program}) :: \left(\begin{array}{c} \textit{clock calculus} \\ \hline \textit{conditional dependence graph} \end{array} \right)$$

Here,

- program denotes the program to be encoded, and ***synch*** is the encoding map;
- *clock calculus* denotes the set of algebraic equations encoding the constraints on synchronisation or logic; these equations can represent either static or dynamical systems as we shall see later;
- *conditional dependence graph* denotes the set of possibly occurring dependencies together with the clocks where these dependencies are in force.

3.1.2.1 Instruction (i): relation or function.

Boolean relation.

The coding of all boolean relations is easily derived from the coding of the following instructions and the coding of the composition we shall see below:

$$\begin{aligned}
&\textit{synch}\,(a := \textit{true})\,:: && \left(\begin{array}{c} a^2 - a = 0 \\ \hline \varnothing \end{array}\right) \\
&\textit{synch}\,(b := \textit{not}\,a)\,:: && \left(\begin{array}{c} b = -a \\ \hline \varnothing \end{array}\right) \\
&\textit{synch}\,(c := a\ \textit{and}\ b)\,:: && \left(\begin{array}{c} a^2 = b^2 \\ c = a^2 - (ab + a + b) \\ \hline \varnothing \end{array}\right)
\end{aligned} \qquad (3\text{-}1)$$

The algebraic equation of the first formula possesses $a = 1, a = 0$ as only solutions, which means that a is either absent or true. The second equation is obvious. To derive the last one, remark that its first component encodes the fact that both signals a and b must have the same clock (they are either both present or absent, which is encoded as $a^2 = 1$ or $a^2 = 0$); then it is straightforward to verify that the last equation maps the pairs (0,0), (1,1), (−1,1), (1,−1), (−1,−1) respectively onto 0, 1, −1, −1, −1. Since only boolean are involved, no coding of dependencies is required hence the symbol ∅ in the second field.

Non boolean function.

$$\textit{synch}\,(y := f(x_1, \ldots, x_n))\,:: \quad \left(\begin{array}{c} y^2 = x_1^2 = \ldots = x_n^2 \\ \hline y^2 : x_1 \longrightarrow y \ \ldots \ x_n \longrightarrow y \end{array}\right) \qquad (3\text{-}2)$$

The first field encodes the constraints on clocks (equality), while the second one encodes the data dependencies. The meaning of the second field is «the listed dependencies hold when $y^2 = 1$». Notice that a := (u<v) produces a boolean, but is a nonboolean function.

3.1.2.2 Instruction (ii): the register.

Boolean register.

This is the key case where dynamical systems in $\mathbf{F}_3$ come out.

$$\textit{synch}\,(b:=a\$\textit{init}\,u)::\left(\begin{array}{c}\xi=(1-a^2)\xi'+a;\ \ \textit{initial cond}=u\\ b=a^2\xi'\\ \hline \varnothing\end{array}\right) \tag{3-3}$$

Here, ξ is the current state of the dynamical system, ξ' its previous state, and u its initial condition (± 1 valued). The corresponding explicit form of this dynamical system is

$$\xi_t=(1-a_t^2)\xi_{t-1}+a_t;\quad \xi_0=u$$
$$b_t=a_t^2\xi_{t-1}$$

where t is any time index fast enough to capture every presence of signal. Notice that the state takes $+1$ or -1 as only values, i.e. states are persistent. The state is modified when a new input is received, and at the same instant the old state is delivered at the output. Again is no dependence graph necessary.

Non boolean register.

$$\textit{synch}\,(y:=x\$\textit{init}\,u)::\left(\begin{array}{c}y^2=x^2\\ \hline \varnothing\end{array}\right) \tag{3-4}$$

The first field expresses that clocks must be identical; the second field is empty even if we are considering non boolean types, since the current value of y does not depend on the current value of x, but on the content of the memory (which has been lost in the coding via ***synch***).

3.1.2.3 Instruction (iii): the filtering.

Filtering with boolean output.

$$\textit{synch}\,(c := b\ \textit{when}\ a) :: \left(\begin{array}{c} c = b(-a - a^2) \\ \hline \varnothing \end{array} \right) \tag{3-5}$$

Filtering with non boolean output.

$$\textit{synch}\,(y := x\ \textit{when}\ a) :: \left(\begin{array}{c} y^2 = x^2(-a - a^2) \\ \hline y^2 : x \longrightarrow y \end{array} \right) \tag{3-6}$$

The second field expresses that x influences y when y is produced.

3.1.2.4 Instruction (iv): the merge.

Merge with boolean output.

$$\textit{synch}\,(c := a\ \textit{default}\ b) :: \left(\begin{array}{c} c = a + b(1 - a^2) \\ \hline \varnothing \end{array} \right) \tag{3-7}$$

Merge with non boolean output.

$$\textit{synch}\,(y := u\ \textit{default}\ v) :: \left(\begin{array}{c} y^2 = u^2 + v^2(1 - u^2) \\ \hline u^2 : u \longrightarrow y \\ v^2(1 - u^2) : v \longrightarrow y \end{array} \right) \tag{3-8}$$

The second field expresses the fact that u influences y when it is present, while v influences y when it is present and u is absent.

3.1.2.5 Instruction (v): the composition.

$$\textit{synch}\,(P \mid Q) = \textit{synch}\,(P) \cup \textit{synch}\,(Q) \tag{3-9}$$

Here the symbol U means that the fields of P and Q have to be merged to produce a single

clock field and a single conditional dependence graph field. This U is in fact a composition in our sense.

Definition: *In the coding above, the first field will be called the* ***clock calculus*** *of the program, while the second one will be called the* ***conditional dependence graph*** *of the program.*

3.1.3 Examples.

To allow drawing of graphs, conditional dependence graphs will be depicted using the following notation:

$$x \xrightarrow{h} y \quad \text{instead of} \quad h{:}x \longrightarrow y$$

3.1.3.1 The instruction *cell*.

This instruction has been used as a macro in the «shared track» example. Recall the corresponding program:

```
y := x cell b init y0
          =
     zy := y$ init y0
     y := x default zy
     synchro y, (x default when(b))
```

We shall distinguish two cases: x boolean, and x non boolean.

Encoding the boolean *cell* into its clock calculus.

$$\xi = (1 - y^2)\xi' + y, \textit{init} = y_0$$
$$zy = y^2\xi'$$
$$y = x + zy(1 - x^2)$$
$$y^2 = x^2 + (-b - b^2)(1 - x^2) \qquad (3\text{-}10)$$

Eliminating zy yields

$$\xi = (1 - x^2)\xi' + x, \textit{init} = y_0$$
$$y = x + (-b - b^2)(1 - x^2)\xi'$$

which reflects exactly the meaning of the instruction ***cell***: the memory is refreshed when x is received, and y delivers the current or last value of x when x is received or b is received and true.

Encoding the non boolean *cell* into its cclock calculus and conditional dependence graph.

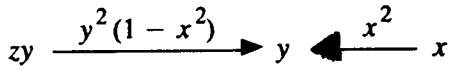
Clock calculus:

$$zy^2 = y^2 = x^2 + (1 - x^2)zy^2$$
$$y^2 = x^2 + (1 - x^2)(-b - b^2)$$

which yields

$$zy^2 = y^2 = x^2 + (1 - x^2)(-b - b^2)$$

Conditional Dependence Graph:



Here the dynamics has been lost; the clock calculus expresses only how clocks of signals are related.

3.1.3.2 The program GUARDED_COUNT.

Recall this program:

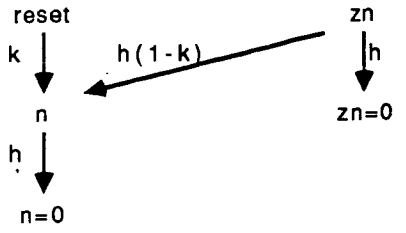
```
GUARDED_COUNT { ? integer reset ! bool empty}
              =
     n = reset default (zn - 1)
   | zn = n$ init 0
   | empty = when(n = 0) default (not event(reset))
   | synchro when(zn = 0), reset

end
```

Clock calculus:

$$n^2 = reset^2 + zn^2(1 - reset^2) = zn^2$$
$$empty = (-[n = 0] - [n = 0]^2) + (-reset^2)(1 + [n = 0] + [n = 0]^2)$$
$$-[zn = 0] - [zn = 0]^2 = reset^2 \qquad (3\text{-}11)$$

Introduce the notations $\alpha = [n = 0]$, $\beta = [zn = 0]$. The clock calculus can be rewritten

$$zn^2 = n^2 = \alpha^2 = \beta^2$$
$$empty = (-\alpha - \alpha^2) + (\beta + \beta^2)(1 + \alpha + \alpha^2)$$
$$reset^2 = -\beta - \beta^2 \qquad (3\text{-}12)$$



$h = n^2$

$k = reset^2$

DISCUSSION: observability.
About the clock calculus: consider the first two equations of (3-11), which corresponds to the first two instructions of the program. Their solution is

$$n^2 = zn^2 = reset^2 + \Phi^2(1 - reset^2)$$

where Φ is a *free* variable of F_3; this additional variable, we shall call a ***phantom***, reflects the fact that the two first instructions of the program are ***not observable*** by the input reset only. The clock of the outputs zn and n of this subprogram is not entirely constrained by the clock of the input reset, which is a cause of nondeterminism of this subprogram. Observability for DEDS is widely discussed in [Ramadge 1986] in a different setting.
However, considering the whole clock calculus yields a different result. This clock calculus is an algebraic variety which is entirely parametrized by the free parameters $\{\alpha, \beta\alpha^2\}$; on the other hand, since the conditional dependence graph has reset as only source node (except from the delay output zn), it is expected that the whole program is observable by the triple {reset, $\alpha, \beta\alpha^2$}. A systematic study of this kind of observability notion is presented in [Benveniste & al. ** 1988].

3.1.3.3 The instruction k = a *fby* k; a control viewpoint.

This instruction has been introduced in the discussion of the shared track example to figure how program synthesis could be performed instead of explicit programming; the actual form of this instruction was

train_i_on := ask_i_fuel *fby* train_i_on

Taking into account the fact that k is a pure event, its expansion is

k := a *fby* k
=
k := k *when* (a *default* zc)
zc := c$ *init false*
c := (*not* k) *default* a

The corresponding clock calculus is

$$\xi = (1 - c^2)\xi' + c, \quad \textit{init} = -1$$
$$zc = c^2\xi'$$
$$c = -k + (1 - k)a$$
$$k = k[-(a + (1 - a^2)zc) - (a + (1 - a^2)zc)^2]$$

which gives after a little algebra

$$\xi = (1 - k)(1 - a^2)\xi' - k + (1 - k)a, \quad \textit{init} = -1$$
$$k(1 - a - (1 - a^2)\xi') = 0$$

which is equivalent to

$$\xi = (1 - k)(1 - a^2)\xi' - k + (1 - k)a, \quad \textit{init} = -1$$
$$k = [-a - a^2 - (1 - a^2)(1 + \xi')]\Phi^2$$

where Φ is a *phantom*, i.e. a free parameter; the presence of this parameter reflects the fact that the input a makes the whole system unobservable (non deterministic in the computer science framework). More precisely, we have

$$\xi' = -1 \quad \Rightarrow \quad k = (-a - a^2)\Phi^2$$

i.e. when a=*true* occurs, k is free but not obliged to occur, otherwise it cannot occur; on the other hand

$$\xi' = 1 \quad \Rightarrow \quad k = (1 + a)^2\Phi^2$$

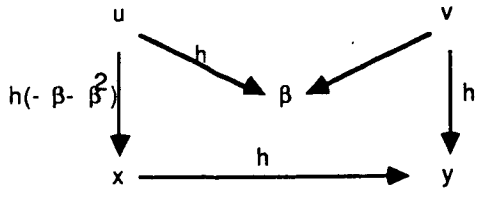
i.e. k is free (but not obliged) to occur provided that a≠*false*. The phantom Φ^2 reflects the degree of freedom left by the program: ***it is in fact the hidden control.*** Here, taking $\Phi^2 = 1$ identically provides the fastest and most frequent response to the boolean signal a. Referring to the shared track example, this is precisely the choice of the controls (one for each train) which provides the same behaviour as the first two programs which did entirely specify the system.

3.1.3.4 An example of deadlock.

Consider the following example.

```
    x := u when (u<v)
|   . y := x + v
```

The meaning of this program is «add u to (v ***when*** u<v)»; this program should be rejected, since the clocks are inconsistent. The conditional dependence graph of this program is



$h = u^2$

Writing β for short instead of (u<v), we get

$$u^2 = v^2 = \beta^2 \quad (i)$$
$$x^2 = u^2(-\beta - \beta^2) \quad (ii)$$
$$y^2 = x^2 = v^2 \quad (iii)$$

which obviously enforces $\beta = 1$. However β is not free, but is the result of the evaluation of the inputs u and v; but neither our clock calculus nor our conditional dependence graph can reason about non-boolean values, so that the actual value (true or false) of β cannot be predicted within our calculus, hence this value should not be constrained. This is taken into account by adding to (ii) the constraint obtained by the symmetry $\beta \rightarrow -\beta$, thus resulting in the new constraint

$$0 = \beta u^2 \quad (ii')$$

instead of (ii), thus yielding finally

$$any^2 = 0$$

i.e. the whole program starvates from the beginning. This illustrates informally how deadlocks can be detected by taking into account clocks and data dependencies. Again this is an illustration of our HDS theory compared to other approaches. A formal study of deadlocks is also presented in [Benveniste & al. ** 1988]; the complete resolution and execution of programs is also shown in this reference.

Chapter Four
Conclusion.

We have presented a brief account of HDS theory, with an illustration on the so-called «shared-track» example relevant to DEDS area, and «Page» example from signal processing, which cannot be captured via DEDS modelling. Notice the following points.

- HDS involve both numerics and symbolics (logic and synchronization), which makes them of wide applicability, but also more difficult to analyse than DEDS; hence our current results are less advanced than the results of DEDS theory.
- Our theory is supported by the programming language SIGNAL. A version V2 of the SIGNAL compiler is currently experimented in some universities, and the language is being soon commercially available. The compiler produces as an intermediate level code the pair {clock calculus, conditional dependence graph}; from this intermediate level code, executable FORTRAN code, but also distributed OCCAM code (programming language of the INMOS "Transputer"), is generated. Current target applications are signal processing systems, radar systems, and a whole continuous speech recognition application. The existence of this programming language makes the treatment of real applications really feasible, cf [Ho 1987].
- Our approach is relational, which results in a simple treatment of exact model following control problems, as well as proof systems such as usually done using temporal logic [Pnueli 1977] (proving a property is in fact an exact model following control problem). The main problem is then HDS resolution, which can be considered as a realization theory (program → state machine executing the program).
- Our theory captures both notions of observability and deadlock, the latter notion being related to controllability.

Finally, let us mention the works around the other declarative (but functional) language LUSTRE [Bergerand & al. 1985] [Capsi & al. 1987] and the imperative language ESTEREL [Berry & Cosserat 1984][Gonthier 1988] the syntax of which possesses some flavour of ADA.

REFERENCES.

[Astrom & al. 1986]: K.J. Astrom, J.J. Anton, K.E. Arzen, «Expert Control», Automatica, 22, 3, 277-286.

[Basseville & Benveniste 1986]: M. Basseville, A. Benveniste Ed. *Detection of Abrupt Changes in Signals and Dynamical Systems*, LNCIS vol 77, Springer V.

[Benveniste & Le Guernic 1987]: A. Benveniste, P. Le Guernic, «A denotational theory of synchronous communicating systems», INRIA research report No 685.

[Benveniste & al. ** 1988]: A. Benveniste, B. Le Goff, P. Le Guernic, «Hybrid Dynamical Systems theory and the language SIGNAL. Part II: mathematical models», INRIA research report.

[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*.

[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.

[Cardelli 1980]: L. Cardelli, «Analog Processes», Proc. of the 9th symposium on Mathematical Foundations of Computer Science, Lect. Notes on Comp. Sc. vol 88, Springer Verlag.

[Cardelli 1982]: L. Cardelli, «Real Time Agents», Proc. of the ninth Colloquium on Automata, Languages and Programming, Aarhus, Denmark, July 1982.

[Caspi et al. 1987]: P. Caspi, D. Pilaud, N. Halbwachs, J.A. Plaice, «LUSTRE: a declarative language for programming synchronous systems», Proc of the 14th ACM symp. on Principles of Programming Languages, 1987.

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

[Dellacherie & Meyer 1976]: C. Dellacherie, P.A. Meyer, *«Probabilites et Potentiel»*, 2nd edition, Hermann, Paris.

[Gautier 1987]: T. Gautier, P. Le Guernic, L. Besnard, «SIGNAL, a Declarative Language for Synchronous Programming of Real-time Systems», proc. of the third Conference on Functional Programming Languages and Computer Architecture, G. Kahn Ed, Lect Notes in Computer Science, vol 274, Springer V.

[Gonthier 1988]: G. Gonthier, thesis, Univ. de Nice and Ecole des Mines, 1988.

[Ho 1987]: Y.C. Ho, «Basic Research, Manufacturing Automation, and putting the Cart before the Horse», Editorial, IEEE-AC-32 No 12, 1042-1043.

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

[Inan & Varaiya 1987]: K. Inan, P. Varaiya, "Finitely Recursive Processes", CDC 1987, 252–256.

[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 & Benveniste 1986]: P. Le Guernic, A. Benveniste, «Real-time synchronous data-flow programming: the language SIGNAL and its mathematical semantics», INRIA research report No 533, reprint as INRIA res. rep. No 620.

[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.

[A. Levis et al. 1987]: «Challenges to Control, a collective view», IEEE–AC–32, 274–285, 1987

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

[Milner 1983]: R. Milner «Calculi for Synchrony and Asynchrony», Theoretical Computer Science, 25 No 3, 267–310.

[Ostroff 1986]: J.S. Ostroff, «Real time computer control of discrete systems modelled by extended state machines: a temporal logic approach», Rep. No 8618, Systems Control Group, Dept of Elec Eng., Univ of Toronto.

[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.

[Ramadge 1986]: P.J. Ramadge, «Observability of discrete event systems», CDC–1986, Athens, 1108–1112.

[Ramadge and Wonham 1987–a]: P.J. Ramadge, W.M. Wonham, «Supervisory Control of a class of Discrete Event Processes», SIAM J. Control and Opt., 25, 1, 1987, 206–230.

[Ramadge and Wonham 1987–b]: P.J. Ramadge, W.M. Wonham, «On the Supremal Controllable Sublanguage of a Given Language», SIAM J. Control and Opt., 25, 3, 1987, 637–659.

[Willems 1987]: J.C. Willems «From time series to linear systems», Automatica

[Young 1982]: S.J. Young, *«Real time languages: design and development»*, Elis Horwood Publishers, 1982.

ACKNOWLEDGEMENT: the authors are indebted to the SIGNAL working group for valuable discussions and permanent exchanges of ideas; they wish to thank M. Sorine and Y. Sorel from INRIA–Rocquencourt, B. Le Goff, T. Gautier and B. Cheron at IRISA, and especially A.S. Willsky and C. Ozveren from MIT who are currently visiting IRISA.

Hybrid Dynamical Systems theory and the language SIGNAL.

Part II: mathematical models.

Albert Benveniste, Bernard Le Goff, Paul Le Guernic
IRISA/INRIA, Campus de Rennes Beaulieu
35042 RENNES CEDEX, FRANCE

Summary: *we study the logic and synchronization characteristics of general dynamical systems called Hybrid Dynamical Systems. Our theory generalizes the notion of Discrete Event Dynamical Systems by handling numerics as well as symbolics. Our theory is supported by the programming language SIGNAL and a mathematical model of relational style. This framework allows us to formulate in the same way HDS programming or specification and HDS control. The core of the theory is the notion of HDS resolution which is based on a reduction technique mapping any HDS specification program into a polynomial dynamical system on the finite field of integers modulo 3. This second part is devoted to the formal presentation of the theory.*

[1] this work is supported by the *accord-cadre CNET-INRIA poste de travail pour le traitement du signal et la conduite de processus*

Table of Contents

Chapter One
Introduction.

In the companion paper [Benveniste & Le Guernic * 1988] we have introduced the reader to HDS theory and presented the language SIGNAL which was designed for HDS specification; we have also discussed the principles of HDS modelling and resolution. The purpose of this paper is to present two models for HDS and a resolution technique supported by these mathematical models.

Before going into the core of the subject, let us recall briefly the mini−language SIGNAL. Here follow its 5 instructions.

p(x(1),...,x(n))	(i)
y := x$ *init* x_0	(ii)
y := x *when* b	(iii)
y := u *default* v	(iv)
P | Q	(v)

Recall their intuitive meaning:

(i): direct extension of instantaneous relations into relations acting on flows:

$$p(x(1),\ldots,x(n)) \Leftrightarrow \forall t : p(x_t(1),\ldots,x_t(n))$$

(ii): logical delay (the usual operator z^{-1})

$$y := x\$ \textit{ init } x_0 \Leftrightarrow \forall t > 1 : y_t = x_{t-1}, y_1 = x_0$$

(iii): condition (b is boolean): y equals x when the signal x and the boolean b are available and b is true; otherwise, y is not delivered;

(iv): y merges u and v, with priority to u when both signals are simultaneously present;

The instructions (i−iv) specify the elementary processes, we shall call *generators*. The objects named x, y, u, v, b, will be called *signals*.

(v): communication of already defined processes: P and Q communicate through their signals with common names; for example

$$y := zy + a \mid zy := x\$ \textit{ init } x_0$$

denotes the system of recurrent equations for $t \geqq 1$

$$y_t = zy_t + a_t$$
$$zy_t = y_{t-1}, zy_1 = x_0$$

which is equivalent to $y_t = y_{t-1} + a_t, y_0 = x_0$. In this example, P and Q are just predefined systems of dynamical equations; but names can be given as well to such systems and used instead inside the instruction (v).

The reader is referred to the companion paper [Benveniste & Le Guernic * 1988] for an introduction to the principles of HDS modelling. All the examples discussed in the sequel are presented in details in [Benveniste & Le Guernic * 1988].

Chapter Two
Hybrid Dynamical System Theory: the approach via Transition Systems.

2.1 The method of Transition Systems.

In his seminal work [Plotkin 1981], G. Plotkin advocated that programming languages should be

- specified
- interpreted
- compiled

using *transition rules*; transition rules are a suitable way to encode and produce the labelled trees which result from the running of a program, as will be explained later on; they can be used to represent finite or infinite state machines. An extensive litterature does exist on this technique to encode languages for communicating processes. Hence we shall first present such a coding for the language SIGNAL introduced above. It is not our purpose to fully develop this approach within the present paper, which will be primarily devoted to the new model called Ω.

2.1.1 Definition of transition systems.

*Definition 3: a **transition system** is a triple of the form*

$$P = \{\Xi, \mathsf{A}, \rightarrow\} \tag{2-1}$$

where

Ξ *is the set of **states**; states are program texts;*

A *is the set of **actions**; actions are functions mapping a given set of ports into a corresponding set of **values**, which are said to be **carried** by these ports during this transition; actions are denoted by*

$$\mathsf{x}(1)(x_1)\ldots\mathsf{x}(n)(x_n)$$

*where the x's are the **ports** of the system and the x's are the values carried by the corresponding ports; among the set of values is the distinguished value $\perp$ which has to be interpreted as the absence of data.*

*$\rightarrow$ is the **transition function**; despite this name, $\rightarrow$ is a **relation** defined on $\Xi \times A \times \Xi$. This transition function is defined by a set of **rules** of the form*

$$C \Rightarrow \quad P \xrightarrow{\alpha} P' \tag{2-2}$$

where P,P' are program texts, i.e. states, α is the considered action, and

$$C = f(P, \alpha, P') \tag{2-3}$$

is a predicate function of the mentioned arguments; the meaning is «P can perform α and yield P' if C holds».

Hence it is clear that performing an action generally requires to solve an implicit equation since the precondition C can depend on the action and on the resulting new state.

Given two actions α and β belonging to (possibly) different transition systems, we introduce the predicate

$$\alpha \cap \beta \tag{2-4}$$

to indicate that the corresponding maps agree on the intersection of their respective domains (which means that ports with identical names carry the same value). When this predicate holds, the two actions are said to be ***compatible***, and their union is defined as usually for maps, keeping only a single copy of the common ports. Union is denoted by $\alpha \cup \beta$.

Transition systems form an algebra endowed with the composition operation defined by

$$\frac{C \Rightarrow \; P \xrightarrow{\alpha} P'; \quad D \Rightarrow \; Q \xrightarrow{\beta} Q'}{(\alpha \cap \beta, C, D) \Rightarrow \quad P \mid Q \xrightarrow{\alpha \cup \beta} P' \mid Q'} \tag{2-5}$$

which has to be read

$$\frac{\text{list of preexisting rules}}{\text{new rule resulting from their combination}}$$

Notice that, when P and Q have no port with common name, all interleaving of actions from P and/or Q is valid, compare with the model Ω we shall introduce in the forthcoming chapter.

2.1.2 Encoding SIGNAL into transition systems.

2.1.2.1 Encoding the instruction (i).

$$p(x_1,\ldots,x_n) \Rightarrow \quad \mathsf{p(x(1),\ldots x(n))} \xrightarrow{\mathsf{x(1)}(x_1)\ldots\mathsf{x(n)}(x_n)} \mathsf{p(x(1),\ldots x(n))} \qquad (2\text{–}6)$$

Here, the states of the transition are just the text of the instruction (there is no memory, hence the state is unchanged). The predicate requires that the values to be presented to the ports x(i) satisfy the relation p; if this holds, then these values can be accepted by the transition, and this acceptance is the action performed by the transition. When this relation is for example z:=x+y, the action has to be interpreted as the computation of the sum of the values carried by the ports x and y and its delivery at the port z. For the encoding of the composition, we shall also need to consider the following trivial rule, which represents the empty action (in which case the program is always unchanged):

$$\mathsf{p(x(1),\ldots x(n))} \xrightarrow{\mathsf{x(1)}(\perp)\ldots\mathsf{x(n)}(\perp)} \mathsf{p(x(1),\ldots x(n))} \qquad (2\text{–}7)$$

All the instructions (ii)–(iv) should be also provided with a corresponding trivial transition, which will thus be omitted.

2.1.2.2 Encoding the instruction (ii).

$$\mathsf{y := x\,\$}\,\textit{init}\,u \xrightarrow{\mathsf{x}(v)\,\mathsf{y}(u)} \mathsf{y := x\,\$}\,\textit{init}\,v \qquad (2\text{–}8)$$

Here, the precondition is the constant *true*, so it is omitted. Here, u is the content of the memory: it is delivered at the output y, while the value v received at the input port x is fed into the memory. Notice that the state has been here modified via the modification of the parameter involved in the program.

2.1.2.3 Encoding the instruction (iii).

$$\begin{aligned}
&\mathsf{y := x}\ \textit{when}\ \mathsf{b} \xrightarrow{\mathsf{x}(x)\ \mathsf{b}(\perp)\ \mathsf{y}(\perp)} \mathsf{y := x}\ \textit{when}\ \mathsf{b} \quad (i)\\
&\mathsf{y := x}\ \textit{when}\ \mathsf{b} \xrightarrow{\mathsf{b}(b)\ \mathsf{x}(\perp)\ \mathsf{y}(\perp)} \mathsf{y := x}\ \textit{when}\ \mathsf{b} \quad (ii)\\
b = \textit{true} \Rightarrow\ &\mathsf{y := x}\ \textit{when}\ \mathsf{b} \xrightarrow{\mathsf{x}(x)\ \mathsf{b}(b)\ \mathsf{y}(x)} \mathsf{y := x}\ \textit{when}\ \mathsf{b} \quad (iii)\\
b = \textit{false} \Rightarrow\ &\mathsf{y := x}\ \textit{when}\ \mathsf{b} \xrightarrow{\mathsf{x}(x)\ \mathsf{b}(b)\ \mathsf{y}(\perp)} \mathsf{y := x}\ \textit{when}\ \mathsf{b} \quad (iv)
\end{aligned} \qquad (2\text{–}9)$$

These four rules encode exactly the intuitive meaning of the instruction ***when***. No change in the state occurs, but the precondition plays a role.

2.1.2.4 Encoding the instruction (iv).

$$\begin{aligned}
&y := u \ \textit{default}\ v \xrightarrow{u(u)\ y(u)\ v(\perp)} y := u \ \textit{default}\ v \quad (i)\\
&y := u \ \textit{default}\ v \xrightarrow{u(u)\ v(v)\ y(u)} y := u \ \textit{default}\ v \quad (ii)\\
&y := u \ \textit{default}\ v \xrightarrow{v(v)\ y(v)\ u(\perp)} y := u \ \textit{default}\ v \quad (iii)
\end{aligned} \qquad (2\text{-}10)$$

This is the exact coding of the intuitive meaning of this instruction. Again, no change in the state occurs, and no precondition is required.

2.1.2.5 Encoding the instruction (v).

This is the major step, since this instruction is the key to modular construction. In fact the corresponding coding has already been given in (2-5).

2.1.3 An example of program coding.

We want to encode the program

```
P(u)        =
    n := reset default xn
    xn := zn - 1
    zn := n$ init n0
```

The rules of the three instructions are the following, together with the trivial rules (2-7)

$$\begin{aligned}
&n := \text{reset}\ \textit{default}\ xn \xrightarrow{\text{reset}(u)\ n(u)\ xn(\perp)} n := \text{reset}\ \textit{default}\ xn \quad (i)\\
&n := \text{reset}\ \textit{default}\ xn \xrightarrow{\text{reset}(u)\ xn(v)\ n(u)} n := \text{reset}\ \textit{default}\ xn \quad (ii)\\
&n := \text{reset}\ \textit{default}\ xn \xrightarrow{xn(v)\ n(v)\ \text{reset}(\perp)} n := \text{reset}\ \textit{default}\ xn \quad (iii)
\end{aligned}$$

$$z = w - 1 \Rightarrow \quad xn := zn - 1 \xrightarrow{zn(w)\ xn(z)} xn := zn - 1$$

$$zn := n\$\ \textit{init}\, n0 \xrightarrow{n(x)\ zn(n0)} zn := n\$\ \textit{init}\, x$$

Combining these rules according to (2-5) yields the set of rules to encode the program P. Choosing (i) requires the precondition

$$\{[\text{reset}(u)\, n(u)\, xn(\perp)] \cap [zn(w)\, xn(z)] \cap [n(x)\, zn(n0)]\} \Leftrightarrow \{u = x, z = \perp, w = \perp, n0 = \perp, x = \perp\}$$

The matching of these actions requires $any = \perp$, i.e. no port participates to the considered action, hence rule (i) cannot be used. The only valid rules are

$$\{v = z, u = x, w = n0, z = w - 1\} \Rightarrow \quad \mathsf{P}(n0) \xrightarrow{\mathsf{reset}(u)\ \ \mathsf{xn}(v)\ \ \mathsf{n}(u)\ \ \mathsf{zn}(n0)} \mathsf{P}(u) \quad (ii)$$

$$\{v = z, v = x, w = n0, z = w - 1\} \Rightarrow \quad \mathsf{P}(n0) \xrightarrow{\mathsf{reset}(\perp)\mathsf{xn}(v)\ \ \mathsf{n}(v)\ \ \mathsf{zn}(n0)} \mathsf{P}(v) \quad (iii)$$

The preconditions of these rules can be simplified by removing the value names which do not appear in the actions of the resulting transitions, thus finally giving (no precondition is required for the first rule)

$$\mathsf{P}(n0) \xrightarrow{\mathsf{reset}(u)\ \ \mathsf{xn}(v)\ \ \mathsf{n}(u)\ \ \mathsf{zn}(n0)} \mathsf{P}(u) \quad (ii)$$

$$\{v = n0 - 1\} \Rightarrow \quad \mathsf{P}(n0) \xrightarrow{\mathsf{reset}(\perp)\mathsf{xn}(v)\ \ \mathsf{n}(v)\ \ \mathsf{zn}(n0)} \mathsf{P}(v) \quad (iii)$$

We have already shown how the algebra of SIGNAL programs can be mapped into the algebra of transition systems. On the other hand, it should be clear from (2-3) that *performing actions requires to solve systems of equations: here comes the need for resolution.* Now, the following problem remains, which is the crux of the theory: ***transform any transition system into a machine which can execute it.*** This problem has a flavour of realization theory: transforming the «external representation» of a transition system (its SIGNAL program, or the corresponding image in the algebra of transition systems) into a «(finite) state machine» realization. This sort of job usually requires a convenient algebraic calculus to be at hand, and it should be clear that there is no hope for us to have this sort of algebra, since our model of transition systems is too general.

Hence the need for a reduction technique: use a convenient homomorphism from the algebra of transition system into itself such that its range be

- small enough to provide some suitable algebraic calculus,
- rich enough to still provide us with a convenient solution to our realization problem.

This will be the subject of the next section.

2.2 The fundamental homomorphism.

2.2.1 Introducing the homomorphism.

Here follows the intuitive description of our method. Recall that SIGNAL consists in extending a given «instantaneous language» (the set of data types and corresponding relations known by the instruction (i)) with a small set of primitives to obtain a model of dynamical systems.

Denoting by IL (Instantaneous Language) this starting point, any homomorphism ψ from IL into itself is easily extended to an homomorphism Ψ from SIGNAL into itself.

In our case, the homomorphism ψ is constructed as follows

STEP 1: among the relations of IL, select the subfamily of relations and corresponding data types for which you accept to solve systems of equations (and are supposed to be able!).

STEP 2: other instantaneous relations must be *functions*, and are encoded into their dependence graph; hence corresponding data types are handled as sets of labels for which dependence graphs summarize all the possible rewritings.

This defines our homomorphism; its ability to reason about dynamical systems as well as its complexity relies on the choice we have done at STEP 1. Here we shall select the boolean variables together with the boolean relations generated by {:=, ***and, or, not***} and the constants *true, false*; this choice is motivated by the particular role played by the booleans in the instruction ***when***. STEP 2 is decomposed as follows

STEP 2.1: mapping data types.

We introduce the set Λ of ***labels*** (or names); the only relation defined on Λ is $a \longrightarrow b$ to denote that a precedes b. Then, we introduce the map ψ mapping a variable of any type into its name. The image of a variable a of any type by ψ will be denoted by $\psi(a)$ or, when no confusion can result, simply by a.

STEP 2.2: extending ψ to mapping instantaneous non-boolean functions into directed graphs.

Non-boolean functions induce branches between their labels as follows

$$\psi: \quad y = f(x_1, \ldots, x_n) \;\Rightarrow\; \{\psi(x_1) \longrightarrow \psi(y), \ldots, \psi(x_n) \longrightarrow \psi(y)\}$$

simply denoted by

$$y = f(x_1, \ldots, x_n) \;\Rightarrow\; \{x_1 \longrightarrow y, \ldots, x_n \longrightarrow y\}$$

ψ extends immediately to systems of equations. The so obtained directed graphs summarize all the possible rewritings such as substituting the label y by $f(x_1, \ldots, x_n)$ in other expressions. In order for such rewritings to terminate properly, it is needed that such graphs generate *partial orders*, i.e. are circuit-free.

This map ψ extends immediately to an homomorphism Ψ from the transition systems algebra into itself: add $\perp$ to the set of labels and booleans and keep unchanged any equation involving $\perp$; this gives an extension of ψ which can be applied to the preconditions and actions of the transition systems. Hence the definition of the ***fundamental homomorphism*** Ψ: it maps the transition (2-2) into the new transition

$$\psi(C) \Rightarrow \quad P \xrightarrow{\psi(\alpha)} P' \tag{2-11}$$

Thanks to the formula

$$\psi\{\alpha \cap \beta, C, C'\} = \{\psi(\alpha) \cap \psi(\beta), \psi(C), \psi(C')\}$$

it is easily seen that Ψ is an homomorphism, i.e. that

$$\Psi(P|Q) = \Psi(P)|\Psi(Q) \tag{2-12}$$

holds.

2.2.2 The image of SIGNAL.

2.2.2.1 Image of the instruction (i).

Boolean relations are unchanged. A non-boolean function is mapped as

$$\begin{array}{c} x_1 \longrightarrow y, \ldots, x_n \longrightarrow y \\ \Downarrow \\ \mathsf{y := f(x(1),...x(n))} \xrightarrow{y\ x_1 \ldots x_n} \mathsf{y := f(x(1),...x(n))} \end{array} \tag{2-13}$$

Here, y, x_i denote the labels.

2.2.2.2 Image of the instruction (ii).

The boolean delay is unchanged. For the nonboolean case, we get

$$\mathsf{y := x\$}\mathit{init}. \xrightarrow{x\ y} \mathsf{y := x\$}\mathit{init}. \tag{2-14}$$

Here the bound on the value carried by the ports via the memory is lost: nonboolean delays are just mapped into pure synchronisation instructions.

2.2.2.3 Image of the instruction (iii).

The *when* with boolean output is unchanged. For the non boolean case, we get

$$\mathsf{y := x}\ \mathit{when}\ \mathsf{b} \xrightarrow{x\ \mathsf{b}(\perp)\ y(\perp)} \mathsf{y := x}\ \mathit{when}\ \mathsf{b} \quad (i)$$

$$\mathsf{y := x}\ \mathit{when}\ \mathsf{b} \xrightarrow{\mathsf{b}(b)\ x(\perp)\ y(\perp)} \mathsf{y := x}\ \mathit{when}\ \mathsf{b} \quad (ii)$$

$$x \longrightarrow y, b = true \Rightarrow \quad y := x \ \textit{when}\ b \xrightarrow{x\ b(b)\ y} y := x \ \textit{when}\ b \quad (iii)$$
$$b = false \Rightarrow \quad y := x \ \textit{when}\ b \xrightarrow{x\ b(b)\ y(\perp)} y := x \ \textit{when}\ b \quad (iv) \qquad (2\text{-}15)$$

These four rules encode exactly the intuitive meaning of the instruction ***when***. No change in the state occurs, but the precondition plays a role.

2.2.2.4 Image of the instruction (iv).

Again, no change for the boolean case; for the nonboolean case, we get

$$u \longrightarrow y \Rightarrow \quad y := u \ \textit{default}\ v \xrightarrow{u\ u\ v(\perp)} y := u \ \textit{default}\ v \quad (i)$$
$$u \longrightarrow y \Rightarrow \quad y := u \ \textit{default}\ v \xrightarrow{u\ v\ y} y := u \ \textit{default}\ v \quad (ii)$$
$$v \longrightarrow y \Rightarrow \quad y := u \ \textit{default}\ v \xrightarrow{v\ y\ u(\perp)} y := u \ \textit{default}\ v \quad (iii) \qquad (2\text{-}16)$$

Finally the image of the instruction (v) has already been defined for Ψ to be an homomorphism. The range of Ψ will be called the algebra of ***synchro-transition systems***, since they summarize the logic, synchronization, and dependency structure of the transition system.

2.2.3 Algebraic representation of synchro-transition systems.

Synchro-transition systems are defined via rules involving $\perp$, booleans, and labels. We shall first provide an algebra with a convenient calculus where the pairs {$\perp$, booleans} can be represented. All we need to encode are the following status: *absent, present, true, false.* These are encoded onto the finite field $\mathbb{F}_3 = \mathbb{Z}/3\mathbb{Z}$ of integers modulo 3 as follows

$$\begin{aligned} &true: +1 \\ &false: -1 \\ &absent: 0 \\ &present: \pm 1 \end{aligned}$$

where ± 1 denotes a non determinate choice of $+1$ or -1; i.e. we handle in the same way labels and boolean of nondeterminate value. Let us apply this idea to encode SIGNAL programs: the corresponding map will be denoted by ***synch***. For this purpose, we need to define precisely what it the domain of our coding, namely the algebra of ***dynamical system*** over $\mathbb{F}_3^n$.

A ***dynamical system*** over $\mathbb{F}_3^n$ is the specification of

1. a submanifold of the product space $\mathbb{F}_3^n \times \mathbb{F}_3^n$;

2. an initial condition in F_3^n.

Indeed, denoting by ξ the generic point of F_3^n, such a submanifold is specified via a system of polynomial equations

$$\begin{aligned} P_1(\xi, \xi') &= 0 \\ &\cdots\cdots \\ &\cdots\cdots \\ P_k(\xi, \xi') &= 0 \end{aligned} \tag{2-17}$$

where $\xi = (x_1, \ldots, x_n)$, and the x_i's are variables in F_3. Then, *the dynamical system is the subset of the trajectories on* F_3^n *satisfying*

$$\begin{aligned} P_1(\xi_t, \xi_{t-1}) &= 0 \\ &\cdots\cdots \\ &\cdots\cdots \\ P_k(\xi_t, \xi_{t-1}) &= 0 \end{aligned} \tag{2-18}$$

where ξ_0 equals the given initial condition.

Using these notions, the algebraic coding of the transition (2–11) is derived using the following rules.

RULE TRANS_1

$$\begin{aligned} x \in \psi(\alpha) &\Rightarrow x^2 = 1 \\ \mathrm{b}(b) \in \psi(\alpha) &\Rightarrow b^2 = 1 \\ x(\perp) \in \psi(\alpha) &\Rightarrow x^2 = 0 \end{aligned}$$

This rule encodes the presence/absence of the values carried by ports within the actions; only squares appear since the value of booleans plays no role here.

RULE TRANS_2

$$\begin{aligned} \{b = \mathit{true}\} \in \psi(C) &\Rightarrow b = 1, \quad \{b = \mathit{false}\} \in \psi(C) \Rightarrow b = -1 \\ \{b = \mathbf{not}\, a\} \in \psi(C) &\Rightarrow b = -a \\ \{c = a \,\mathbf{and}\, b\} \in \psi(C) &\Rightarrow c = 1 - (ab + a + b) \end{aligned}$$

This rule concerns the instantaneous language only. It suffices to encode the constraints on the booleans within the preconditions. Only the last part needs to be verified by checking all the combinations of ± 1 values for a and b.

RULE TRANS_3

$$\{x \longrightarrow y\} \in \psi(C) \Rightarrow x \longrightarrow y$$

In other words, dependencies are kept unchanged in this coding.

Let us illustrate how the coding works on the transitions of the instruction ***when*** in the non boolean case; starting from the transitions (2-15), we get

$$\begin{array}{c} x^2 = 1, b^2 = 0, y^2 = 0 \quad (i) \\ b^2 = 1, x^2 = 0, y^2 = 0 \quad (ii) \\ b = 1, x^2 = 1, y^2 = 1, x \longrightarrow y \quad (iii) \\ b = -1, x^2 = 1, y^2 = 0 \quad (iv) \end{array} \tag{2-19}$$

Since any one of these transitions can be applied, these equations can be summarized as the double coding

$$\textit{synch}\,(y := x \textit{ when } a) :: \left(\begin{array}{c} y^2 = x^2(-a - a^2) \\ \hline y^2 : x \longrightarrow y \end{array} \right) \tag{2-20}$$

In the second field of this coding, «y^2:» is a shortage to indicate that the dependency holds exactly when $y^2 = 1$. A systematic application of this method yields exactly the coding of the SIGNAL instructions which was given in the section (3.1.2) of [Benveniste & Le Guernic * 1988]; this coding is not repeated here.

2.3 Conclusion of the chapter.

We have presented a model of HDS using transition systems. We have shown how general transition systems can be mapped into the subalgebra of synchro−transition systems. Finally we have exhibited an algebraic coding of these transition systems via pairs {clock calculus, conditional dependence graph}. What remains to be done is to provide an algorithm to *solve* such systems. However we shall first present the second point of view, namely the model Ω.

Chapter Three
A new relational model: Ω.

The basic approach of Ω is to consider that the mathematical model of a SIGNAL program should be a formal way to define the set of the behaviours of its signals via constraints on the set of all possible behaviours. Hence, the aim of model Ω is to capture in a single framework the following questions:

- how to formalize the notion of *space of all possible behaviours?*
- how to define *constraints* on such spaces?
- how to build a theory of *multiple clocked systems?*
- how to build a *hierarchical* theory of multiple clocked systems, i.e. to let predefined subsystems to be composed to construct an overall system?
- how to analyze properties of the so-obtained dynamical systems (observability, deadlock −a property related to controllability−).

To achieve this, the background of Ω model is the so−called technique of *canonical measurable spaces* of probability theory [Dellacherie and Meyer 1976]. For those who are familiar with probability theory, let us recall the following tools from probability theory

- canonical spaces of trajectories, to describe the set of all possible behaviours;
- families of σ−algebras, to describe the flow of time;
- *stopping times*, to describe events that can be observed while monitoring the process;
- *stacks*, from Ornstein's ergodic theory, to describe oversampling

Unfortunately, no classical tool does exist to combine these objects to obtain a hierarchical theory. We shall here present a slightly simplified version of the Ω model; a complete presentation can be found in [Benveniste & Le Guernic 1987].

3.1 Histories.

3.1.1 Basic definitions: histories, signals, clocks.

3.1.1.1 Histories.

In the sequel, the symbols $\mathbf{N}, \mathbf{N}_+$ denote respectively the ordered sets $\{0,1,2,\ldots,\infty\}$ and $\{1,2,\ldots,\infty\}$. By ∞, we have in mind an instant for which one waits for ever, i.e. an event occurring at ∞ never happens.
The notion of history we shall introduce now is borrowed from [Kahn 1974], and is also in accordance with the corresponding name which is sometimes used by probabilists to refer to increasing families of σ-algebras, as the following example will show.

A basic example: history associated to the instruction (i).

Consider the instruction

$$y = f(x(1),\ldots,x(n))$$

We shall write X for short to denote the n-uple (x(1),...,x(n)). Introduce the function set

$$\Omega = \{\mathbf{N}_+ \rightarrow \Xi \times Y\} \tag{3-1}$$

where Ξ denotes the set in which the signal X takes its values, Y is the set where the output y takes its values, and denote by ω the elements of Ω: every ω represents thus a possible trajectory $\xi_1, \xi_2, \ldots$ of the input/output pair (X,y). Then, endow Ω with the following equivalence relation

$$\{\omega \approx_t \omega'\} \Leftrightarrow \{\omega(s) = \omega(s') \ \forall s \leqq t\} \tag{3-2}$$

Let us define a partition Π_t as follows: ω and ω' belong to the same atom of Π_t iff $\omega \approx_t \omega'$. By convention, $\Pi_0 = \{\varnothing, \Omega\}$ represents the information available before the time starts (constants,...), which does not depend on the values of the inputs. Then $\{\Omega, (\Pi_t)_{t \in \mathbf{N}}\}$ is said to be the *canonical history associated to the instruction (i)*. The partition Π_t is interpreted as *the information available at time t*, i.e. any data available at time t in the system is nothing but a function defined on the set Ω, which is constant on the atoms of the partition Π_t, in other words a function of the initial segment $(X,y)_{[1,t]}$. We shall now introduce the notion of history in an abstract setting to generalize this example.

***Definition 4:** By an history, we have in mind an object*

$$\{\Omega, (\Pi_t)_{t \in \mathbf{N}}\} \quad \text{or} \quad \{\Omega, \Pi\} \quad \text{for short}$$

where

- *Ω is a set*
- *for every t, Π_t is a partition of the set Ω; the elements of Π_t are referred to as **atoms***
- *for $s<t$, Π_t is **finer** than Π_s, denoted by $\Pi_s \leqq \Pi_t$, i.e. every atom of Π_s is a union of atoms of Π_t.*

The family of partitions (Π_t) *is called the* ***information flow*** *of the history.*

The partition Π_t has to be interpreted as «the information available at time t».

Notations.

1/ Given a function X defined on Ω, we shall write for short

$$X \in \Pi_t \tag{3-3}$$

to indicate that X is constant on the atoms of the partition Π_t; in the same way, if A is a subset of Ω, we shall write

$$A \in \Pi_t \tag{3-4}$$

to indicate that A is a union of atoms of the partition Π_t; these notations will be extended to the other partitions we shall encounter in the sequel.
2/ Given a logical proposition $P(\omega)$ depending on ω, we shall write for short

$$P \text{ instead of } \{\omega : P(\omega)\} \tag{3-5}$$

For example, $\{X > \lambda\}$ denotes the set $\{\omega : X(\omega) > \lambda\}$.

3.1.1.2 Clocks.

A clock relates any sequence of events to a given time reference, by specifying at which date the s−th event occurs.

Definition 5: *Given a history* $\{\Omega, (\Pi_t)\}$, *a* $\{\Omega, (\Pi_t)\}$-**clock** H *(or, simply a clock, when there is no ambiguity) is a time-indexed family of functions*

$$H_t : \ \Omega \to N \tag{3-6}$$

which satisfies the following properties $\forall \omega$

$$\begin{aligned} &H_0(\omega) = 0, \quad H_\infty(\omega) = \infty && (i) \\ &s < t \quad \text{and} \quad H_s(\omega) < \infty \ \Rightarrow \ H_s(\omega) < H_t(\omega) && (ii) \\ &H_t(\omega) \leqq s \quad \text{and} \quad \omega' \approx_s \omega \ \Rightarrow \ H_t(\omega') = H_t(\omega) && (iii) \end{aligned} \tag{3-7}$$

Recall that $H_s(\omega) = \infty$ means that the s−th event of the clock H never happens. The last condition is a causality condition: it expresses the fact that, to know if the t-th occurrence of an event $H(\omega)$ takes time not after s, it is sufficient to observe the initial segment up to time s.

Counters. Counters are associated to clocks as follows. Given a $\{\Omega,(\Pi_t)\}$-clock H, the following formula, where # stands for short for «cardinal of»,

$$\mu_t^H(\omega) = \#\{s \in \mathbf{N} : H_s(\omega) \leqq t\} \tag{3-8}$$

defines the *counter associated to H.* Thanks to (3-7-iii), counters enjoy the following property:

Lemma 1: we have

$$\{\omega' \approx_t \omega\} \Rightarrow \{\mu_t^H(\omega') = \mu_t^H(\omega)\}$$

Traces. The notion of trace we shall now introduce is in fact the proper framework to handle simultaneously both dual aspects of time, namely clock/counter.

Definition 6: *the trace associated to a clock H is the subset* $\overline{H}$ *of* $\Omega \times N$ *defined by*

$$(\omega, s) \in \overline{H} \Leftrightarrow \exists t : H_t(\omega) = s \tag{3-9}$$

It will be convenient to introduce the following notations:

$$\begin{aligned} \overline{H}(\omega, .) &= \{s \in N : (\omega, s) \in \overline{H}\} \\ \overline{H}(., s) &= \{\omega \in \Omega : (\omega, s) \in \overline{H}\} \end{aligned} \tag{3-10}$$

The interest of this notion is that the clock and the counter can be easily recovered from it, as we shall explain now.

$$\begin{aligned} \mu_t^H(\omega) &= \#\{\overline{H}(\omega, .) \cap [0, t]\} \\ H_t(\omega) &= \min\left\{s : \#\{\overline{H}(\omega, .) \cap [0, s]\} = t\right\} \end{aligned} \tag{3-11}$$

Moreover, we have the following result:

Lemma 2: Given a subset $\overline{H}$ of $\Omega \times N$, the formulas (3-11) define a clock iff the following property holds

$$\forall s \leqq t : \overline{H}(., s) \in \Pi_t \tag{3-12}$$

Example.

Consider a causal signal X, which means in our framework that

$$\omega' \approx_t \omega \Rightarrow X_t(\omega) = X_t(\omega') \tag{3-13}$$

and let A be a subset of the domain in which X takes its values. Define

$$\begin{aligned} H_0(\omega) &= 0 \\ H_1(\omega) &= \min\{s > 0 : X_s(\omega) \in A\} \\ H_{t+1}(\omega) &= \min\{s > H_t(\omega) : X_s(\omega) \in A\} \end{aligned} \tag{3-14}$$

where, by convention, $\min(\varnothing) = +\infty$. Then, H is a clock, for it satisfies the constraint (3-7-iii) thanks to (3-13).

This example can also be written in the mini language SIGNAL; we assume for example that X takes real values, and that $A =]\lambda, \infty[$. The corresponding instruction is

H = *true* **when** X>λ

The set of the occurrences of the signal H defines exactly the trace of the clock H introduced in (3-14).

History associated to a clock. This notion will be a first step towards the technique of time change. The idea supporting time changes is the following: suppose you have a clock, and you are interested in an infinite ordered file of data which are available at each occurrence of this clock. Then, you would probably like to forget the original time reference, and prefer to work with the above mentioned clock *as it were the time reference.* Our aim is to justify such a procedure. A first step in this direction is the following definition.

Definition 7: *Let H be a $\{\Omega, (\Pi_t)\}$-clock. Define*

$$\omega \approx_{H_t} \omega' \quad \Leftrightarrow \quad \exists s:\ s = H_t(\omega) \text{ and } \omega \approx_s \omega' \tag{3-15}$$

This is an equivalence relation; (3-15) defines (Π_{H_t}) as the ***history associated to the clock*** *H.*

The fact that (3-15) defines an equivalence relation is due to the property (3-7-iii) of time-correctness of the clock H. The corresponding information flow summarizes the flow of information corresponding to every new occurrence of H, a fundamental notion when the study of time changes is of interest. This notion is the good functional point of view to encode the notion of initial segment in the case of time changes.

3.1.1.3 Causal signals.

Definition 8: *Let H be a $\{\Omega, (\Pi_t)\}$-clock. By a $\{\Omega, (\Pi_t), H\}$-signal* (for short instead of «causal signal»), *we have in mind a time-indexed family $X = (X_t)_{t \in \mathbf{N}_+}$ of functions*

$$X_t : \Omega \to \Xi$$

(where Ξ is a set defining the type of the signal), *satisfying the following* ***time-correctness*** *condition:*

$$\omega' \approx_{H_t} \omega \quad \Rightarrow \quad X_t(\omega') = X_t(\omega) \tag{3-16}$$

The definition 8 expresses the fact that $X_t(\omega)$ has to be considered as known at time $H_t(\omega)$; in this case, $H_t = \infty$ means that X_t is never delivered.

3.1.2 The clock algebra.

The aim of this section is to study the algebra of $\{\Omega, (\Pi_t)\}$-clocks, we shall simply refer to as «clocks», since no confusion can occur throughout this chapter. We shall introduce a useful primitive operation on this set: the *filtering*.

3.1.2.1 A partial order on the set of the clocks.

It is defined as follows. Given two clocks H and K, referring to (3–10), we define

$$K \subset H \Leftrightarrow \forall \omega : \overline{K}(\omega, .) \subset \overline{H}(\omega, .) \tag{3-17}$$

In other words, $K \subset H$ means that the set of occurrences of K is included in the set of the occurrences of K *whatever the input* ω *is*, i.e. $\subset$ is a partial order on functions. Hence the set of clocks with $\subset$ is a lattice with $\varnothing$ (the clock with no event at all) as minimal element and *Id* (the identity clock) as maximal element. The operations of infimum and supremum will be respectively denoted by

$$K \wedge H, K \vee H \tag{3-18}$$

3.1.2.2 The filtering.

Every signal of boolean type will be said to be a *test*. The following lemma holds, where the notation 1_A yields 1 when the property A holds, and 0 otherwise:

Lemma 3: Let H be a clock, and T be a $\{\Omega, (\Pi_t), H\}$-test. Then, the following formula

$$K_t(\omega) = \min \left\{ H_s(\omega) : \sum_{u=1}^{s} 1_{\{T_u(\omega) = true\}} \geq t \right\} \tag{3-19}$$

defines a new clock, denoted by

$$K = H \downarrow T \tag{3-20}$$

and referred to as the clock obtained by filtering H by T.

Proof: easy and left to the reader.

The meaning of the filtering is as follows: $H \downarrow T$ *extracts from* H *the instants* H_s *where* T_s *is true.* Conversely, the following result holds, where we have used the notation (3–10):

Lemma 4: If $K \subset H$ in the sense of (3–17), then $K = H \downarrow T$ holds, where the $\{\Omega, (\Pi_t), H\}$-test T is given by

$$T_s(\omega) = \begin{cases} \textit{true} \text{ if } H_s(\omega) \in \overline{K}(\omega, .) \\ \textit{false} \text{ otherwise} \end{cases} \tag{3-21}$$

The proof is elementary, and is left to the reader.

3.1.3 History communication and time changes.

To define history communication, we shall proceed as follows. First, we define the *history juxtaposition*, which is the result of observing two histories which do not communicate at all. Then, a simple restriction operation will provides us with the abstract notion of *history communication*.

3.1.3.1 History juxtaposition.

The idea behind history juxtaposition is simple. An observer which monitors two histories $\Omega\Pi = \{\Omega, \Pi\}$ and $\Omega\Pi' = \{\Omega', \Pi'\}$ that do not exchange any information will observe the interleaving of the events belonging to $\Omega\Pi$ or $\Omega\Pi'$. We shall now formulate this idea rigorously by proceeding into two steps.

STEP 1: inserting dummy events.

Consider a history $\Omega\Pi = \{\Omega, \Pi\}$. Introduce the following objects.

$$\perp = \{\mathbf{N} \to \mathbf{N}\} \tag{3-22}$$

Elements of $\perp$ are denoted by γ. Notice that γ is allowed to take the value ∞. Introduce

$$\Omega^{\perp} = \Omega \times \perp \tag{3-23}$$

COMMENT: The intepretation of $\Omega^{\perp}$ is as follows: elements of this set are of the form (ω, γ), where γ is a given sequence of nonnegative integers; $\gamma(t)$ specifies how many «dummy» events (i.e. events to which $\Omega\Pi$ do not participate) will be inserted between the instants t and $t + 1$ of the history $\Omega\Pi$. Notice that *inserting* ∞ *dummy events means that* $\Omega\Pi$ *is waiting for ever for its next event, i.e.* $\Omega\Pi$ *starvates from now on.*

Then, $\Omega^{\perp}$ will be endowed with the following information flow:

$$(\omega, \gamma) \approx_t (\omega', \gamma') \tag{3-24}$$

$$\Leftrightarrow$$

$$u_0 = \max\{u: \sum_{v=0}^{u} (1 + \gamma(v)) \leq t\} \Rightarrow \begin{cases} \forall u \leq u_0 : \gamma'(u) = \gamma(u) \\ \omega \approx_{u_0} \omega' \end{cases}$$

The figure below depicts two trajectories of $\Omega^{\perp}$ that are equivalent in the above introduced sense. This technique for inserting additional instants between preexisting ones originates from D. Ornstein's method to prove the isomorphism theorem between Bernoulli dynamical systems in ergodic theory [Ornstein 1974].

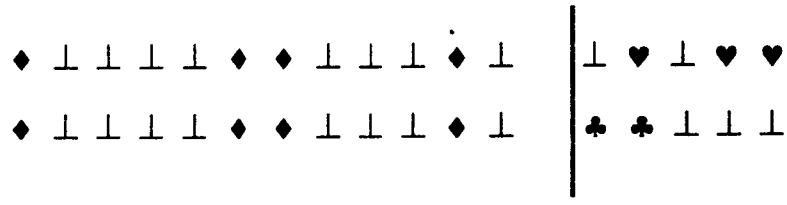


Figure 1: two trajectories equivalent up to $t = 12$

The corresponding information flow will be denoted by $(\Pi_t^{\perp})$.

STEP 2: defining history juxtaposition.

Definition 9: *Given two histories* $\Omega\Pi = \{\Omega, \Pi\}$ *and* $\Omega\Pi' = \{\Omega', \Pi'\}$, *their* ***juxtaposition*** *is denoted by*

$$W\Gamma = \Omega\Pi \;\&\; \Omega\Pi' \tag{3-25}$$

and is defined as follows.
STEP 2.1: consider the cartesian product

$$\{\Omega^{\perp} \times \Omega'^{\perp}, \Pi_t^{\perp} \times \Pi_t'^{\perp}\}$$

STEP 2.2: consider the clock H^{Ω} *of the events in which* $\Omega\Pi$ *is involved:*

$$\overline{H}^{\Omega} = \{(w,t) \in W \times \mathbf{N}:\ \exists u,\ \sum_{v=0}^{u} (1 + \gamma(v)) = t\}$$
$$\text{where } w = ((\omega, \gamma), (\omega', \gamma')) \tag{3-26}$$

and define similarly $\overline{H}^{\Omega'}$. *STEP 2.3: delete the trajectories which possess at least one instant in which neither* $\Omega\Pi$ *nor* $\Omega\Pi'$ *has an event:*

$$W\Gamma = \{W, \Gamma\}, \quad \text{where}$$
$$W = \{w \in \Omega^{\perp} \times \Omega'^{\perp}:\ \overline{H}^{\Omega}(w,.) \cup \overline{H}^{\Omega'}(w,.) = \mathbf{N}\}$$
$$\Gamma_t = \{\Pi_t^{\perp} \times \Pi'^{\perp}_t\}_{|W} \tag{3-27}$$

The fact that H^{Ω} is a clock is easy to verify, using lemma 2 and (3–24).

Property: *The operation & is commutative and associative.*

Here, the product of partitions is defined in an obvious way,

$$\{\bullet\}_{|W}$$

denotes the restriction of $\{\bullet\}$ to W. The definition of W simply means that we keep in W the subset of $\Omega^{\perp} \times \Omega'^{\perp}$ composed by the trajectories all the events of which belong either to the history $\Omega\Pi$ or to the history $\Omega\Pi'$. The history Γ corresponds to the flow of information which is received by a simultaneous observer of $\Omega\Pi$ and $\Omega\Pi'$. A picture of this construction is shown below

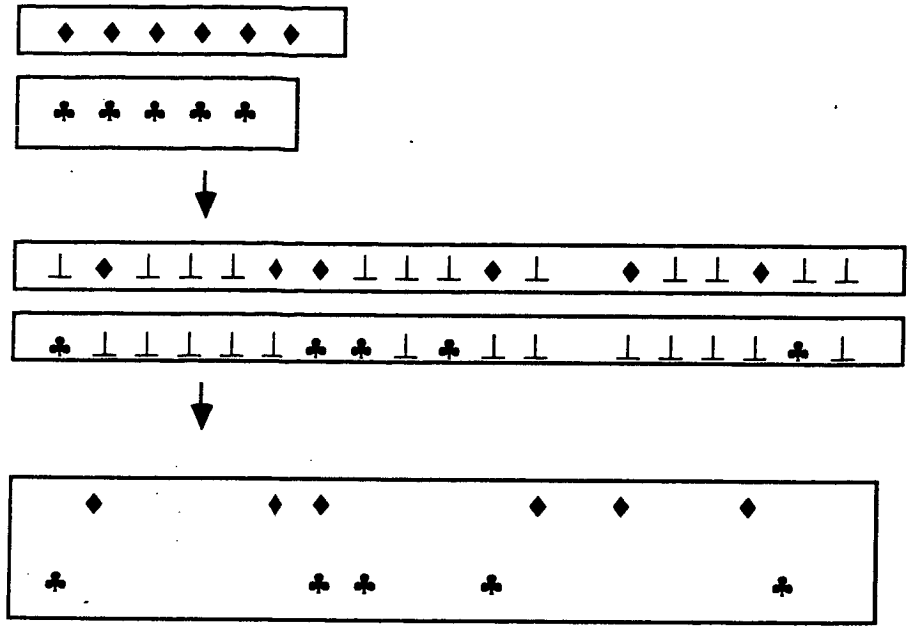


Figure 2: history juxtaposition

Here all the possible interleavings of the two considered histories have been constructed, compare with the model of transition systems for a communication between two transition systems with no common ports.

3.1.3.2 History communication.

Definition 10: *A **communication** of* $\Omega\Pi$ *and* $\Omega\Pi'$ *is defined through the specification of a subset of the trajectory space W of the juxtaposition* $\Omega\Pi \,\&\, \Omega\Pi'$. *The history* Γ *is then restricted accordingly. The set of these history communications will be denoted by*

$$\Omega\Pi \mid \Omega\Pi' \tag{3-28}$$

For example, $\Omega\Pi \,\&\, \Omega\Pi'$ is a particular «communication» of $\Omega\Pi$ and $\Omega\Pi'$, namely the weakest one, in which $\Omega\Pi$ and $\Omega\Pi'$ do not communicate at all.

3.1.3.3 Embeddings.

The aim of this section is to answer the following question: referring to the notations of the preceding section, *is there any natural way to embed the clock (resp. signal) algebra of* $\Omega\Pi$ *into the corresponding algebras of* $W\Gamma$*?* A satisfactory answer to this question is a key pass to time changes.

Theorem 1:

(i) Let H be a $\{\Omega, \Pi\}$-clock. Denote by

$$H = Id \downarrow T \tag{3-29}$$

its decomposition according to Lemma 4. Then, the formula

$$H^{\perp} = H^{\Omega} \downarrow T^{\perp} \tag{3-30}$$

(where the operation $.^{\perp}$ on signals is defined in (3-31) below) *defines a $\{W, \Gamma\}$-clock we shall call the **embedding of H into** $W\Gamma$.*

(ii) Let X be a $\{\Omega, \Pi, H\}$-signal. The formula

$$X_t^{\perp}(\omega, \gamma) = X_t(\omega) \tag{3-31}$$

*defines a $\{W, \Gamma, H^{\perp}\}$-signal we shall call the **embedding of X into** $W\Gamma$.*

Proof: first prove (ii) when $H = Id$, then (i) follows, and finally (ii) in the general case.

Notation: when there is no ambiguity, we shall no more distinguish between signals or clocks and their respective embeddings, hence we shall drop the superscript $\bullet^{\perp}$ when referring to these embeddings.

3.1.4 Observers and observability.

3.1.4.1 Example.

Consider the program:

```
u = zu + 1
zu = raz default past_u
past_u = u$ init u_0
```

As one can guess (this will be formally verified later), the information flow Γ of the history associated to this program corresponds to *all the events to which anyone of the signals* u, zu, raz, *participates.* But it turns out that this program possesses only the signal raz as input, which generates an information flow which is *strictly smaller* than Γ in an intuitive sense (which will be formalized later). This discrepancy between information flows is reflected in the fact that *the input-output "map"* raz → (u,zu) *is in fact not a map:* this happens since the amount of events between two successive occurrences of raz is not specified by the observation of raz itself.

This is exactly how the paradox about «how synchronous deterministic process communication can generate nondeterminism» was introduced in [Brock & Ackerman 1981]. We shall now further investigate this point.

3.1.4.2 Observability, and observer's viewpoint.

Definition 11: Consider a history $\{\Omega,\Pi\}$, *and denote by* clock($\{\Omega,\Pi\}$) *the algebra of its clocks.*

(i) An information flow (Γ_t) *on* Ω *is said to be an* ***observer*** *of* $\{\Omega,\Pi\}$ *if*

$$\exists H \in \text{clock}(\{\Omega,\Pi\}): \quad \Gamma_t \subseteq \Pi_{H_t} \tag{3-32}$$

(ii) Given an observer Γ, *if the following condition holds*

$$\left\{\forall\omega, \forall s: H_t(\omega) \leqq s < H_{t+1}(\omega)\right\} \Rightarrow \left\{\omega' \approx_{\Gamma_t} \omega \Rightarrow \omega' \approx_{\Pi_s} \omega\right\} \tag{3-33}$$

then, the history $\{\Omega,\Pi\}$ *is said to be* ***observable with respect to the observer*** Γ, *while it is said to be* ***unobservable*** *otherwise.*

In (3-33), the relation $\approx_{\Gamma_t}$ means that the considered trajectories belong to the same atom of the partition Γ_t.

COMMENT: The condition (i) expresses that what an observer knows about the considered history is contained in the set of the flashes on this history at the events of a given clock. The condition (ii) expresses that the information flows Γ and Π give the same information about the considered history. A discrepancy between both information flows mean that there exists clocks or signals on this history which cannot be separated by the considered observer: this is the notion of nondeterminism introduced in [Brock and Ackerman 1981]. As the example above showed, this situation can occur even in simple cases. The interest of this rather abstract notion of observability is that it will lead very easily to effective criteria to recognize it.

3.1.5 History congruence.

As we shall see later, it is interesting to identify histories which cannnot be distinguished by any observer; such histories will be said to be congruent. Before defining congruence, we shall need the stronger notion of isomorphism.

3.1.5.1 Isomorphisms.

Definition 12: Two histories $\{\Omega,\Pi\}$ *and* $\{\Omega',\Pi'\}$ *are said to be* ***isomorphic*** *if there exists a bijection* $\Phi:\Omega\to\Omega'$ *such that* $\Phi(\Pi_t)=\Pi'_t \forall t$.

3.1.5.2 Congruence.

We shall need the following notation:

$$\{\Omega, \Pi\}/\Pi_\infty \tag{3-34}$$

will denote the history obtained by taking the quotient set Ω/Π_∞ as trajectory space, and by modifying the information flow Π accordingly.

Definition 13: *Two histories* $\{\Omega, \Pi\}$ *and* $\{\Omega', \Pi'\}$ *are said to be* ***congruent****, denoted by*

$$\{\Omega, \Pi\} \sim \{\Omega', \Pi'\} \tag{3-35}$$

if the quotient histories $\{\Omega, \Pi\}/\Pi_\infty$ *and* $\{\Omega', \Pi'\}/\Pi'_\infty$ *are isomorphic.*

The name of «congruence» is justified by the fact that the operations of history juxtaposition and history communication are compatible with the relation $\sim$. Notice that, in our theory, there is a unique notion which covers both the notion of *trace congruence* and *bisimulation* which are different in the classical asynchronous models of communicating processes [Bloom & al. 1987].

CONCLUSION about the model of histories. We have first introduced the notion of history to investigate what can be built when the set of the available stimuli is fixed. We have analysed the algebras of clocks and of signals. The basic operation of filtering was introduced, and it has been proved in [Benveniste & Le Guernic 1987, theorem 2 and section 4.3.1] that filtering, together with history communication, are the convenient primitives to construct all the clocks of a given history. Then, we have introduced the notion of history communication, and derived associated embeddings for the clocks and signals of the original histories. Finally, we have introduced an elegant definition of observability.

But our definition of history communication is by no means effective, since we never said how the subset W in the definition 10 should be constructed. There are several possible ways to specify systematic rules to construct W: the purpose of the next chapter is to present one of these, which is closely related to the language SIGNAL. The model of processes will be now introduced without illustrations via examples. The reason is that, first we hope the reader to be now more familiar with the technique, and second we shall use this model of processes to build a model of the language SIGNAL.

3.2 Processes.

3.2.1 Definition of processes.

Definition 14: A process is a triple

$$P = \{\Omega, \Pi, \mathbf{A}\} \tag{3-36}$$

here, $\{\Omega, \Pi\}$ *is an history, and* $\mathbf{A}$ *is a finite collection of* ***ports****, i.e. triples of the form*

$$[a, X, H] \tag{3-37}$$

where "a" is a name, H a clock, and X a signal with clock H. We shall say that the ***port*** *named "a" carries the signal X with clock H.*

We have obviously in mind that process communication (we shall define next) can occur through ports only. Referring to this terminology, the names occurring in the instructions of the mini-languages SL_i are port names, and these instructions specify relationships between the signals carried by these ports, and between their respective clocks. Processes will be graphically represented as follows

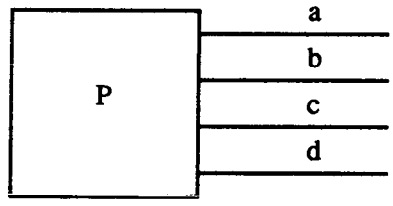


Figure 3: graphical representation of processes.

3.2.2 Process communication.

Definition 15: Assume two processes $P = \{\Omega, \Pi, \mathbf{A}\}$ *and* $P' = \{\Omega'; \Pi'; \mathbf{A}'\}$ *are given; denote by* $\mathbf{B}$ *and* $\mathbf{B}'$ *respectively the subsets of the members of* $\mathbf{A}$ *and* $\mathbf{A}'$ *which possess the same name in both processes. Then the* ***communication*** *of P and P' is denoted by*

$$Q = P \mid P' \tag{3-38}$$

and is defined as follows, referring to the notations of definition 10 and theorem 1:

$$Q = \{W, \Gamma, \mathbf{C}\}, \tag{3-39}$$

where W is the subset of the trajectories of $\{\Omega,\Pi\}\&\{\Omega',\Pi'\}$ *such that*

$$\forall[b,X,H]\in\mathbf{B},\ \forall[b,X',H']\in\mathbf{B}':\ \begin{cases}X'_t(w)=X_t(w)\\ H'_t(w)=H_t(w)\end{cases} \tag{3-40}$$

and

$$\mathbf{C}=\mathbf{A}\cup(\mathbf{A}'-\mathbf{B}')$$

Property: *Process communication is commutative and associative.*

Here, we have used the sketchy notations to refer to embedded clocks and signals, i.e. we have dropped the superscripts $\bullet^{\perp}$. Finally, we identify the connected ports in the resulting process. The whole procedure is graphically depicted as follows

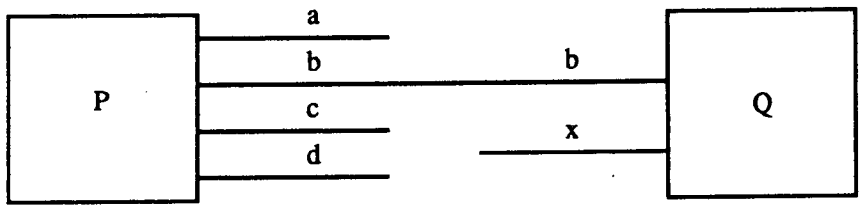


Figure 4: process communication.

COMMENT: This definition expresses that processes do communicate via ports with the same name. This communication constraints the clocks of the corresponding carried signals to be identical, and requires these signals to be equal. Notice that W is never void, since it contains at least w's with infinitely many inserted dummy events (in this case requiring a communication causes a deadlock).

COMMENT: Let us emphazise that communicating via ports with identical names is only a possible way among many others to specify process communication. For instance, in our case, process communication results in a set of constraints between the involved ports, i.e. our model is a *relational* one. It should have also been possible to distinguish «input» and «output» ports, and to allow only «input ↔ input» or «output → input» connections. Such a technique is required if the purpose is to build a functional model; this point will be discussed further in the sequel.

3.2.3 History generated by a family of ports.

Consider a process $P = \{\Omega, \Pi, \mathbf{A}\}$, and a subset **B** of **A**. We shall denote by

$$\sigma(\mathbf{B}) \tag{3-41}$$

the smallest information flow with respect to which all elements of **B** be respectively signals and clocks. Such a minimal information flow does exist, and is the infimum of the set of the sub-information flows of Π such that all elements of **B** satisfy the usual causality properties.

3.2.4 Process congruence.

Following [Milner 1982] and [Austry & Boudol 1983], we introduce a notion of equivalence with respect to a set of observable ports. The subsequent notions are immediate extensions of the corresponding ones for histories.

Definition 16:

(i): Two processes $P = \{\Omega, \Pi, \mathbf{A}\}$ *and* $P' = \{\Omega', \Pi', \mathbf{A}'\}$ *are said to be* ***isomorphic*** *if there exist bijections* $\Psi: \mathbf{A} \to \mathbf{A}'$ *and* $\Phi: \Omega \to \Omega'$ *such that*

$$\forall [a, X, H] \in \mathbf{A}: \begin{cases} \Phi(\Pi_t) = \Pi'_t \\ (\Psi(X))_t(\Phi(\omega)) = X_t(\omega) \\ (\Psi(H))_t(\Phi(\omega)) = H_t(\omega) \end{cases} \tag{3-42}$$

where $\Psi(X), \Psi(H)$ *denote respectively the signal and clock of* $\Psi(\mathbf{A})$.

(ii): Two processes $P = \{\Omega, \Pi, \mathbf{A}\}$ *and* $P' = \{\Omega', \Pi', \mathbf{A}'\}$ *are said to be* ***congruent,*** *denoted by* $P \sim P'$ *if the quotient processes* $P/\sigma(\mathbf{A})$ *and* $P'/\sigma(\mathbf{A}')$ *are isomorphic, where*

$$P/\sigma(A) := \{\Omega/\sigma(\mathbf{A})_\infty, \sigma(\mathbf{A}), \mathbf{A}\}$$

Notice that inserted dummy events, as introduced for the definition of history juxtaposition, are ignored by the observation criterion corresponding to our congruence $\sim$; this compares with the non observation of delays in the weak bisimulation of SCCS, [Milner 1982]. The congruence is compatible with process communication, i.e.

$$P \sim P' \text{and} Q \sim Q' \Rightarrow P|Q \sim P'|Q'$$

As a consequence, in the sequel we shall no more distinguish congruent processes. Notice that restricting the observer to a smaller one ($\mathbf{B} \subset \mathbf{A}$) would cause the loss of the compatibility property; this is a wellknown situation. We are now ready to introduce the model of SIGNAL

programs, i.e. to construct the first homomorphism we did announce from the algebra of SIGNAL programs into the algebra of processes.

3.3 The model of SIGNAL programs.

3.3.1 Notations.

3.3.1.1 The mapping.

The desired mapping will be denoted as follows

$$\mathbf{M}[\text{program}] \;::\; \left(\begin{array}{c} \mathbf{EPROC}[\text{program}] \\ \hline \mathbf{SYNCH}[\text{program}] \\ \hline \mathbf{VAL}[\text{program}] \end{array}\right)$$

where

- «program» denotes a SIGNAL program, i.e. text.
- **M[ ]** denotes the desired mapping, the result of which is a process in the sense of the definition 14; the process **M[program]** is entirely defined by the three fields specified on the right handside.
- the field **EPROC** denotes a process which is an extension of **M[program]**; the convenient restriction is specified by the two next fields.
- the field **SYNCH** is a specification of the relations between the clocks of the ports of **M[program]**.
- the field **VAL** is a specification of the constraints on the various signals (in the sense of the definition 8) involved in **M[program]**.

To summarize, the style of the semantics is the following. First, we specify an extension of the desired process; second we specify the convenient restriction of the space Ω of this extension, and this is obtained via the specification of constraints on signals and clocks of the ports involved in the extended process.

3.3.1.2 Notations for the field «EPROC»

Canonical process associated to a signal. Given a signal x of type Ξ involved in a program, we introduce

$$\mathbf{M}[\mathrm{x}] = \{\Omega^{\Xi},\ o([\mathrm{x},X,Id]),\ [\mathrm{x},X,Id]\}$$

$$\Omega^{\Xi} = \{\mathbf{N} \to \Xi\}$$

$$\omega \in \Omega^{\Xi}:\ X_t(\omega) = \omega(t) \qquad (3\text{-}43)$$

M[x] is a process we shall call the *canonical process of* x. This process summarizes what can be known when observing x only.

Use of process communication. We shall also make use of the process communication P | Q as defined in the definition 15.

3.3.1.3 Notations for the field «SYNCH».

Relationships between clocks will be simply specified by a set of relations written in terms of the operations of the clock algebra, namely the three operations

$$\vee,\ \wedge,\ \downarrow$$

respectively defined in (3-18), (3-20). Given two such sets **SYNCH1** and **SYNCH2**, we shall denote by **SYNCH1** ∪ **SYNCH2** the union of these sets, where identical names refer to the same objects. Finally, given a port $[b,X,H]$, we shall often write generically $\bar{H}(\bar{b})$ to denote the clock of the port named b.

3.3.1.4 Notations for the field «VAL».

We have to specify relationships between the values of different signals at every instant. Such relations will be specified in the following way. Given two ports $[b,X,H]$ and $[b',X',H']$, we shall write formulas of the form

$$K:\ f(X,X') \qquad (3\text{-}44)$$

In this formula

- K denotes a clock, or an expression written in terms of the clock algebra, satisfying the condition $K \subset H \wedge H'$

- $f(x,y)$ denotes a relation.

The meaning of (3-44) is the following (cf. (3-9) for the definition of traces):

$$\forall(\omega,u)\in\overline{K}:\ \begin{cases} H_t(\omega)=u \\ H'_s(\omega)=u \end{cases}$$

$$\Downarrow$$

$$f(X_t(\omega),X'_s(\omega)) \qquad (3\text{-}45)$$

A finite set of such formulas will be sufficient to specify entirely the desired constraints on the values of signals. As for the preceding field, the notation **VAL1 U VAL2** denotes the union of the referred sets of relations on values. Finally, given a port named *b*, we shall generically denote by the same name «b» the signal of this port. We are now ready to present the semantics of SIGNAL.

3.3.2 The model of SIGNAL.

3.3.2.1 Encoding the instruction (i).

$$\mathbf{M[p(x(1),\ldots,x(n))]} \ ::\ \left(\begin{array}{c} \mathbf{M[x(1)]}|\ldots.|\mathbf{M[x(n)]} \\ \hline H(\mathrm{x}(1))=\ldots=H(\mathrm{x}(n)) \\ \hline H(\mathrm{x(i)}):\ p(x(1),\ldots,x(n)) \end{array}\right)$$

We choosed as extended process the communication of the canonical processes of the involved ports; in this case, communication is equivalent to juxtaposition. All signals have the same clock. When the signals are present, their values must satisfy the specified relation.

3.3.2.2 Encoding the instruction (ii).

$$\mathbf{M}[y := x \$ init\, u] \quad :: \quad \left(\begin{array}{c} \mathbf{M[y]\ bv\ M[x]} \\ \hline H(\mathrm{y}) = H(\mathrm{x}) \\ \hline H(\mathrm{x}) : y_1 = u;\ \forall t > 1 : y_t = x_{t-1} \end{array} \right)$$

3.3.2.3 Encoding the instruction (iii).

$$\mathbf{M}[\mathrm{y} := \mathrm{x}\ when\ \mathrm{b}] \quad :: \quad \left(\begin{array}{c} \mathbf{M[b]\,|\,M[x]\,|\,M[y]} \\ \hline H(\mathrm{y}) = H(\mathrm{x}) \wedge (H(\mathrm{b}) \downarrow \mathrm{b}) \\ \hline H(\mathrm{y}) : y = x \end{array} \right)$$

The notation $\downarrow$ refers to the filtering of a clock by a test, see (3-20). The third field expresses that, if y is present, it carries the value of x.

3.3.2.4 Encoding the instruction (iv).

$$\mathbf{M}[\mathrm{y} := \mathrm{u}\ default\ \mathrm{v}] \quad :: \quad \left(\begin{array}{c} \mathbf{M[u]\,|\,M[v]\,|\,M[y]} \\ \hline H(\mathrm{y}) = H(\mathrm{u}) \vee H(\mathrm{v}) \\ \hline H(\mathrm{u}) : y = u;\ \ [H(\mathrm{v}) - (H(\mathrm{u})) \wedge H(\mathrm{v}))] : y = v \end{array} \right)$$

3.3.2.5 Encoding the instruction (v).

$$\mathbf{M}[\mathrm{P}|\mathrm{Q}] \quad :: \quad \left(\begin{array}{c} \mathbf{M[P]\,|\,M[Q]} \\ \hline \mathbf{SYNCH[P] \cup SYNCH[Q]} \\ \hline \mathbf{VAL[P] \cup VAL[Q]} \end{array} \right)$$

In fact, only the first field is necessary to give the entire semantics, thanks to the definition 15 of the process communication. We have nevertheless written all the fields explicitely for the sake of clarity.

Another equivalent semantics of the composition is the following:

$$\mathbf{M[P|Q]} \;::\; \left(\begin{array}{c} \mathbf{EPROC[P]|EPROC[Q]} \\ \hline \mathbf{SYNCH[P] \cup SYNCH[Q]} \\ \hline \mathbf{VAL[P] \cup VAL[Q]} \end{array}\right)$$

Here, EPROC denotes *any extension of the mentioned process.* In this case, the specification of the other fields is necessary. This alternative modelling can be sometimes more convenient.

3.4 The fundamental homomorphism.

3.4.1 Introducing the homomorphism.

We shall use the same method as used before for transition systems to project Ω into a smaller subalgebra. We refer the reader to the discussion of STEPS 1 and 2 followed for the former construction of ψ. Then the map Ψ is defined as follows.

If the process P is described in a normal form as follows

$$P:: \left(\begin{array}{c} \mathbf{M}[a_1]|\ldots|\mathbf{M}[a_n] \\ \hline R_k(H_{a_1},\ldots,H_{a_n}); k=1,\ldots,K \\ \hline R'_i(H_{a_1},\ldots,H_{a_n}) : P_i(X_{a_1},\ldots,X_{a_n}) \text{ where } P_i \text{ is bool} \\ R'_j(H_{a_1},\ldots,H_{a_n}) : Y = F_j(X_1,\ldots,X_n); F_j \text{ function} \end{array}\right) \qquad (3\text{–}46)$$

then its image by Ψ is given by the normal representation

$$\Psi(P)::\left(\begin{array}{c} \mathbf{M}[a_1]|\ldots|\mathbf{M}[a_n] \\ \hline R_k(H_{a_1},\ldots,H_{a_n});k=1,\ldots,K \\ \hline R'_i(H_{a_1},\ldots,H_{a_n}):P_i(X_{a_1},\ldots,X_{a_n});P_i\,\mathrm{bool} \\ R'_j(H_{a_1},\ldots,H_{a_n}):X_1\longrightarrow Y,\ldots,X_n\longrightarrow Y \end{array}\right) \qquad (3\text{-}47)$$

Recall that, in $\Psi(P)$, ports are either boolean or labels. This means that canonical trajectory processes $\mathbf{M}[a]$ for labels are in fact reduced to the single trajectory $\omega=(a,a,a,a,\ldots)$; the associated information flow delivers only the time spent from the origin.

Processes such as $\Psi(P)$ will be called ***synchro-processes*** in the sequel, since data types playing a central role in the synchronisation mechanisms (namely boolean) are preserved. The following theorem is an immediate consequence of the definition of process communication:

Theorem 2: *The map Ψ is an homomorphism from the process algebra Ω into itself, i.e.*

$$\Psi(P|Q)=\Psi(P)|\Psi(Q)$$

Consequently, we get another homomorphism by considering the map ***synch*** defined as follows:

$$\textit{synch}:\ \{\text{algebra of SIGNAL programs}\}\xrightarrow{M}\Omega\xrightarrow{\Psi}\Psi(\Omega)$$

The SIGNAL compiler will make use of the homomorphism obtained via this composition of maps.

3.4.2 Algebraic representation of synchro-processes.

The same method used before for transition systems can also be used here to get an algebraic representation of synchro-processes, with the same results (hence we do not recall them). Let us however point out the following facts.

- The generic form (3-47) of synchro-processes is already equational, unlike the corresponding coding via transition systems.
- Here the field **EPROC** can be understood (hence deleted) since there is only a single data type to be considered, namely the booleans; thus the first field is standard once the list of ports is given.

For these reasons, deriving the algebraic coding from (3–47) is indeed immediate, as shown in the example of the nonboolean ***when*** below:

$$\Psi(y := x \textit{ when } a) :: \left(\begin{array}{c} \mathbf{M}[a] \,|\, \mathbf{M}[x] \,|\, \mathbf{M}[y] \\ \hline H(y) = H(x) \wedge (H(a) \downarrow a) \\ \hline H(y) : x \longrightarrow y \end{array} \right)$$

$$\textit{synch}\,(y := x \textit{ when } a) :: \left(\begin{array}{c} y^2 = x^2(-a - a^2) \\ \hline y^2 : x \longrightarrow y \end{array} \right)$$

We refer again the reader to [Benveniste & Le Guernic * 1988, section 3.1.2] for the complete algebraic coding of SIGNAL.

3.5 A criterion for observability.

We shall say that a process $P = \{\Omega, \Pi, \mathbf{A}\}$ is ***totally observed*** if, according to (3–41),

$$\Pi = \sigma(\mathbf{A}) \tag{3–48}$$

This means that the process is observable by its ports.

Theorem 3: Let $P = \{\Omega, \Pi, \mathbf{A}\}$ *be a totally observed* (3–48) *process, and let* $\mathbf{C} \subset \mathbf{A}$ *be a subset of ports of P* (**C** is intended to refer to a subset of «visible» ports of P), *and denote by* **B** *the subset of* **A** *composed by the boolean ports involved in a non boolean relation. Then, (i)* ⇒ *(ii), where*

(i): this process is observable by the observer $\sigma(\mathbf{C})$

(ii): its static clock calculus satisfies the following condition

(DET): every point in **STATCLOCK[*P*]** *is entirely determined by its components in* $\mathbf{C} \cup \mathbf{B}$.

COMMENT: Here, «points» refers to the points of the algebraic variety defined by the set of equations of **STATCLOCK[*P*]**. **STATCLOCK[*P*]** is the clock calculus obtained by deleting in

the clock calculus of P the state equation of all boolean registers: hence boolean registers are encoded as the single output equation

$$\textit{synch}(\mathsf{b} := \mathsf{a\$}\ \textit{init}\ \mathsf{u}) :: \left(\begin{array}{c} b = a^2\xi' \\ \hline \varnothing \end{array}\right) \qquad (3\text{–}49)$$

instead of the equation (3−3), section (3.1.2) of [Benveniste & Le Guernic * 1988].

COMMENT: the theorem expresses that unobservability in a process can be detected using the clock calculus; notice that the clock calculus can only provide a *sufficient* condition for a process to unobservable by a given subset of its ports. When these ports are intput ports, this turns out to make the process «nondeterministic» (exhibiting different admissible behaviours for a single input history).

Proof: easy, left to the reader; it is not possible to cancel the set **B**, since, while the values of the corresponding signals are well defined by the observation of **C** in a deterministic process, these values can be unconstrained in **STATCLOCK[*P*]**. This criterion has been widely used in the analysis of the examples in [Benveniste & Le Guernic * 1988].

3.6 Conclusion of the chapter.

At this point we have presented two models for the algebra of SIGNAL programs. We have first sketched the model of transition systems, which represents the classical approach of computer science to code languages for communicating processes. Then we presented in details a new model, called Ω, which corresponds to a fully different point of view. Looking at this model from the viewpoint of the computer scientist, this model is of denotational style, the prototype of which is the *Dynamic Network Process* model of Kahn and Mc Queen [Kahn 1974][Kahn & Mc Queen 1977]. However, processes *do relate* histories, i.e. our model is relational unlike DNP. But it should be clear from the preceding chapter that, even if only functions are accepted in the instruction (i) of the language, process composition gives raise to *implicit* systems of equations, i.e. our approach is of deep relational nature. For this reason, we decided to start from the beginning with a fully relational viewpoint. Notice that a similar point of view is strongly advocated by J.C. Willems [Willems 1987] in the control community for the definition of linear systems. It is our opinion that the approach of Ω is cleaner than the approach via transition systems, since all the aspects of the language are encoded within a single framework: constraints on sets of trajectories.

Chapter Four
HDS resolution.

In the preceding section, we showed how the pair {clock calculus, conditional dependence graph} can be used to encode and analyse a SIGNAL program, and more generally, any process of Ω (or any transition system); the image of this coding is the subalgebra of *synchro-processes*. In particular, it was shown how to recognize unobservability. The purpose of this section is to investigate the following questions about synchro-processes:

- what is deadlock and how to detect it;
- construct the machine which can execute synchro-processes.

The latter point corresponds to solving synchro-processes.

4.1 Solving static clock calculi: a first account.

Static clock calculi are clock calculi which do not involve boolean states (or memories), which means that the rule (3-49) has been used to encode the boolean delays; obviously, the algebraic variety defined by the static clock calculus is just the projection of the clock calculus (more precisely the orbits of the dynamical system defined by the clock calculus) along the time axis. Static clock calculi are the crux to the solution of synchro-processes. They are of the generic form

$$\begin{aligned} P_1(X_1,\ldots,X_n) &= 0 \\ &\ldots\ldots\ldots \\ P_K(X_1,\ldots,X_n) &= 0 \end{aligned}$$

where the X_i's are the variables of the static clock calculus, and the P_k's are polynomials of $F_3[X_1,\ldots,X_n]$ of degree at most 2 with respect to each variable.

Definition 17: a static clock calculus is said to be ***pre-solved*** *if it is composed by equations of one of the following forms:*

$$\begin{aligned} Y &= AX^2 + BX + C, \quad A,B,C \text{ polynomials free from the variable } X \quad (i) \\ Y^2 &= AX^2 + BX + C, \quad A,B,C \text{ polynomials free from the variable } X \quad (ii) \end{aligned} \qquad (4\text{-}1)$$

such that every variable is

- *either absent from the left handside of all equations,*

- *or appearing once at the left handside in only one of the two forms (i) and (ii) above.*

The form (i) means that the value of Y is bound, while the form (ii) means that only the clock of the variable Y is bound, while its actual value ($+1$ or -1) might be free (this is the case for boolean variables which are produced by non boolean functions such as b := (x<y) or for labels originating from non-boolean signals). Notice that cycles of mutually defined variables can exist in pre-solved clock calculi.

As usually in computational algebraic geometry [Buchberger 1970, 1979], obtaining a pre-solved form is performed via elimination techniques. The basic lemma for elimination is

Lemma 5:

$$aX^2 + c = 0 \Leftrightarrow \begin{cases} c(a+c) = 0 \\ X^2 = \Phi^2(1-a^2) - ac \end{cases} \tag{4-2}$$

$$aX^2 + bX + c = 0$$

$$\Leftrightarrow$$

$$\begin{cases} c[(a+c)^2 - b^2] = 0 \\ X = \Phi[\prod_{y=a,b,c}(1-y^2)] - (1-a^2)bc + a[b + (1+\Phi^2)(b^2 - ac)] \end{cases} \tag{4-3}$$

In both equations, Φ denotes a phantom, i.e. an additional free variable.

COMMENT: the first rule is convenient to solve for clocks, while the second one has to be used for boolean relations. Both rules have the form

$$\text{equation} \Leftrightarrow \begin{cases} \text{adding constraints on the remaining variables when } X \text{ is eliminated} \\ \text{defining } X \text{ or } X^2 \text{ in terms of the other variables} \end{cases}$$

Proof: the proof rests on the following formulas that are useful and immediate

$$\begin{aligned} &\{p = 0 \text{ and } q = 0\} \Leftrightarrow \{p^2 + q^2 = 0\} \\ &\{p = 0 \Rightarrow q = 0\} \Leftrightarrow \{q(1-p^2) = 0\} \end{aligned} \tag{4-4}$$

We prove only (4-3), since (4-2) is easier and follows the same lines. For the considered equation to have a solution, the following constraints must be satisfied by the triple $\{a, b, c\}$:

$$\{a = b = 0\} \Rightarrow \{c = 0\}$$
$$\{a \neq 0\} \Rightarrow \{\Delta^2 = b^2 - ac \neq -1\}$$

Notice that Δ is nothing but the discriminant of the equation. Combining these constraints using (4–4) yields the constraint in (4–3). Then the definition of X follows easily as usually in college algebra; notice that in this second equation, $1 + \Phi^2$ is a writing of $\pm$; the same phantom can be used in the two terms of the definition of X since these two terms are used when $a = 0$ and $a \neq 0$ respectively.

Using lemma 5 allows to perform elimination when an ordering of the equations and variables has been chosen; a detailed algorithm will be presented later. synchro-processes with pre-solved clock calculi will be called *pre-solved synchro-processes.* In the sequel of this chapter, we shall write *clock calculus* for short to refer to the *static* clock calculus.

4.2 The graph of a pre-solved synchro-process.

The main difficulty in solving synchro-processes is due to the presence of two different kinds of ordering, namely

- the ordering of the variables and of the equations required for the elimination to be performed,
- the ordering resulting from the conditional dependence graph.

Both orderings interact. In fact, elimination must be performed by taking into account data dependencies, and moreover *it is not possible to know a clock depending on the value of a boolean signal resulting from a non boolean function prior to evaluating this function;* on the other hand evaluating such functions require to know their clock. This interaction makes the solution of HDS much more difficult to solve than the solution of processes involving only boolean and logic, but no numerical computations, such as usually studied as DEDS (Discrete Event Dynamical Systems) by control scientists. The purpose of this paragraph is to introduce the main tool to handle this interaction.

Definition 18: *The graph of a synchro-process is the graph obtained by considering branches of the form*

$$X \xrightarrow{P} Y \qquad (4\text{–}5)$$

where P is a clock encoded by its polynomial expression in F_3*, and X and Y are either variables of the clock calculus or labels of the conditional dependence graph.*

The intuitive meaning is: «X influences Y when P=1». We make use of the following conventions to simplify graphs:

RULE GRAPH_0

$$\{X \xrightarrow{P} Y \text{ and } P = 0\} \Rightarrow \{\text{delete the branch}\}$$
$$\{X \xrightarrow{P} Y \text{ and } X \text{ absent}\} \Rightarrow \{\text{delete the branch}\}$$
$$\{X \xrightarrow{P} Y \text{ and } P = 1\} \Rightarrow \{X \longrightarrow Y\}$$

The graph is built according to the following rules, where **A** denotes the set of the labels of the conditional dependence graph. Elements of **A** are written in boldface, and the corresponding clock is written with the same letter in italics, e.g. **a** and a^2. Finally, CDG denotes the conditional dependence graph introduced before, and G denotes the graph of the synchro-process we build now.

RULE GRAPH_1

$$\{\mathbf{a} \in \mathbf{A}\} \Rightarrow \{a^2 \xrightarrow{a^2} \mathbf{a}\} \in G \tag{4-6}$$

To have access to the value of a nonboolean signal, we need to know whether it present or not at the considered instant, i.e. we need to know its clock.

RULE GRAPH_2

$$\{P : \mathbf{x} \longrightarrow \mathbf{y}\} \in CDG \Rightarrow \left\{ \begin{array}{l} \mathbf{x} \xrightarrow{P} \mathbf{y} \\ P \xrightarrow{P} \mathbf{y} \end{array} \right\} \in G \tag{4-7}$$

The first part of the rule is the exact translation of the contribution of CDG; the second part expresses that to evaluate y, we must know when the considered dependency holds, i.e. we must know the actual value of P.

RULE GRAPH_3

$$\{Y \text{ or } Y^2 = AX^2 + BX + C \in \textbf{STATCLOCK}\} \Rightarrow \left\{ \begin{array}{c} X \xrightarrow{B^2} Y \text{ or } Y^2 \\ X^2 \xrightarrow{A^2(1-B^2)} Y \text{ or } Y^2 \end{array} \right\} \tag{4-8}$$

X influences Y when $B \neq 0$, while only X^2 influences Y when $B = 0$ and $A \neq 0$.

RULE GRAPH_4

This rule will be used for synchro-process which are not pre-solved; for these processes the rule

GRAPH_3 does not cover all the cases. For any clock equation to which RULE GRAPH_3 does not apply (it is not of the form $YorY^2 = \ldots$, or more than a single expression is available on the right handside of $YorY^2 = \ldots$)

$$AX^2 + BX + C = 0,\ A,B,C\text{free}\ \text{from}X \Rightarrow \left\{ \begin{array}{c} X \xrightarrow{B^2} any \\ X^2 \xrightarrow{A^2(1-B^2)} any \end{array} \right\} \qquad (4\text{–}9)$$

where *any* refers to any variable of the considered equation except X.

COMMENT: Up to now, we were unable to derive mathematically these rules from the Ω model; they are currently justified by the algorithm we shall propose to execute synchro-processes and the fundamental theorem we shall prove in the next section.

Example: the program GUARDED_COUNT
Using the synchro-process encoding this program (cf the paragraph (3.1.3.2) of the companion paper [Benveniste & Le Guernic * 1988]), we get

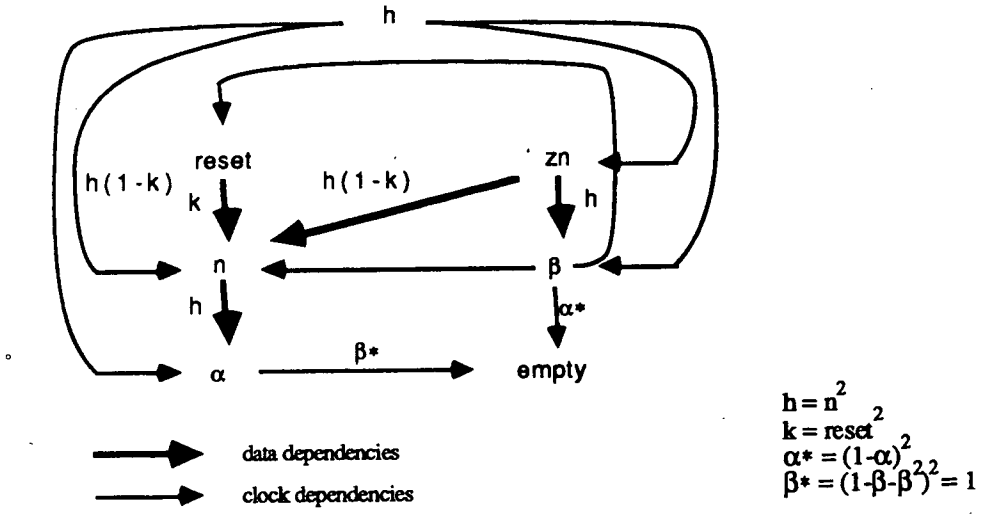


Figure 5: the graph of the program GUARDED_COUNT

IMPORTANT REMARK: It should be clear from the rules GRAPH_3 and GRAPH_4 that the graph of a synchro-process is by no means left invariant by transformations of the clock calculus which preserve the underlying algebraic variety: two isomorphic synchro-processes can have different associated graphs. This property will be exploited in the sequel

We are now ready to present the algorithm EXEC for the execution of synchro-processes; this algorithm is used at run-time.

4.3 Synchro-process execution: the algorithm EXEC.

The algorithm is described by the propagation of tokens along the graph G and the static clock calculus CLOCK. The presence of a token indicates that the corresponding node of the graph G or variable of the clock calculus has been evaluated. The algorithm can be implemented as an automaton, so that we shall describe it using transition systems.

4.3.1 Introducing the notations for EXEC.

4.3.1.1 States.

STATES of such transition systems will be a pairs {G, CLOCK} of graph and clock calculus. The clock calculus is partitioned according to

$$CLOCK = \mathit{val}(CLOCK) \cup ?(CLOCK)$$

i.e. variables which have been evaluated (and replaced by their value) and the others.
Similarly the nodes of G are partitioned according to

$$nodeG = \mathit{val}(nodeG) \cup \mathit{absent}(nodeG) \cup ?(nodeG)$$

where ***absent***(...) refers to the nodes which are known to be absent in the considered step.

4.3.1.2 Actions.

ACTIONS of these transition systems will be elements of the following list

$$\text{CLOCK : list of } \langle\!\langle X \leftarrow \mathit{val}(X) \rangle\!\rangle \quad (i)$$
$$\text{G : list of } \langle\!\langle x \leftarrow \mathit{val}(x) \rangle\!\rangle \quad (ii)$$

The first rule means «the algebraic variable X is substituted by its value in CLOCK»; then all variables of CLOCK which can now be evaluated have to be; finally all the so evaluated clock variables are substituted by their values on the branches and nodes of G. If this actual value is 0, the considered branch is said to be ***broken***.
The second rule means «the label x is substituted by its value in G»; notice that this action concerns the evaluation of nonboolean functions only. Then if x turns out to be boolean (remember x =(u<v)!)), its value is substituted for the corresponding variable in CLOCK and in the clock nodes of G. In the forthcoming rules, we shall omit for short the mention

«CLOCK:» or «G:» in the actions (i) and (ii): this mention is understood according to the type of the node which is being evaluated (label i.e. non-boolean, or element of CLOCK).

4.3.1.3 Preconditions.

PRECONDITIONS of these transition systems are of the form:

$$X \in source(G) \quad (i)$$
$$X \in val(nodeG) \quad (ii)$$
$$P \in val(CLOCK) \quad (iii)$$
$$X \xrightarrow{P} Y \quad (iv)$$

(i) indicates that X is a source node of G; (ii,iii,iv) have already been defined. When no confusion can occur or when this information is unnecessary, the mention (nodeG) or (CLOCK) will be omitted in the preconditions (ii) and (iii).

4.3.2 The rules of EXEC.

Here follow the rules which describe the algorithm.

RULE EXEC_0

$$X \in source(G) \Rightarrow \{G, CLOCK\} \xrightarrow{\text{CLOCK}: X \leftarrow val(X)} \{G, CLOCK : X \in val(CLOCK)\}$$

The source nodes of G are immediately evaluable at the beginning of any instant, and their values are substituted for the corresponding variables in CLOCK; notice that source nodes are always elements of CLOCK. Notice that this rule asssumes that *the list of the source nodes involved in the considered instant is known*; if the considered process communicates with the external world it is implicitely assumed that the external world provides this information.

RULE EXEC_1

$$\forall \{X \xrightarrow{P} Y\} \in CDG:\ X \in val(G) \cup absent(G) \text{ and } P \in val(CLOCK)$$
$$\Downarrow$$
$$\{G, CLOCK\} \xrightarrow{Y \leftarrow val(Y)} \{G : Y \in val(G), CLOCK\}$$

If Y is such that every incoming branch of G has been evaluated, then Y can be evaluated; notice that a possible result of this evaluation is $Y absent$. No change results in the clock calculus from the use of this rule.

RULE EXEC_2

$$X \in \{ val(G) \cap ?(CLOCK) \}$$
$$\Downarrow$$
$$\{G, CLOCK\} \xrightarrow{\text{CLOCK}: X \leftarrow val(X)} \{G, CLOCK : X \in val(X)\}$$

When a new boolean has been evaluated as the result of a non boolean function, then its value is substituted in the clock calculus and all the clocks which can now be evaluated are evaluated. This rule does not modify the graph.

COMMENTS:
1/ Of course, it is desired that EXEC be an algorithm where *preconditions are explicit* unlike general transition systems as pointed out before. We shall give in the next section a sufficient condition to ensure this property.
2/ Notice that EXEC is a *fully parallel algorithm,* i.e. all the actions which can be performed at a given step are performed in parallel; on the other hand, any sequential folding of EXEC can be realized as an automaton using standard realization theory for regular languages.

Example: the program GUARDED_COUNT.
Here follow one possible run of this program, based on its graph we have shown before. In depicting the execution, we make use of the simplification rule GRAPH_0. Nodes which are being evaluated are marked by a token.

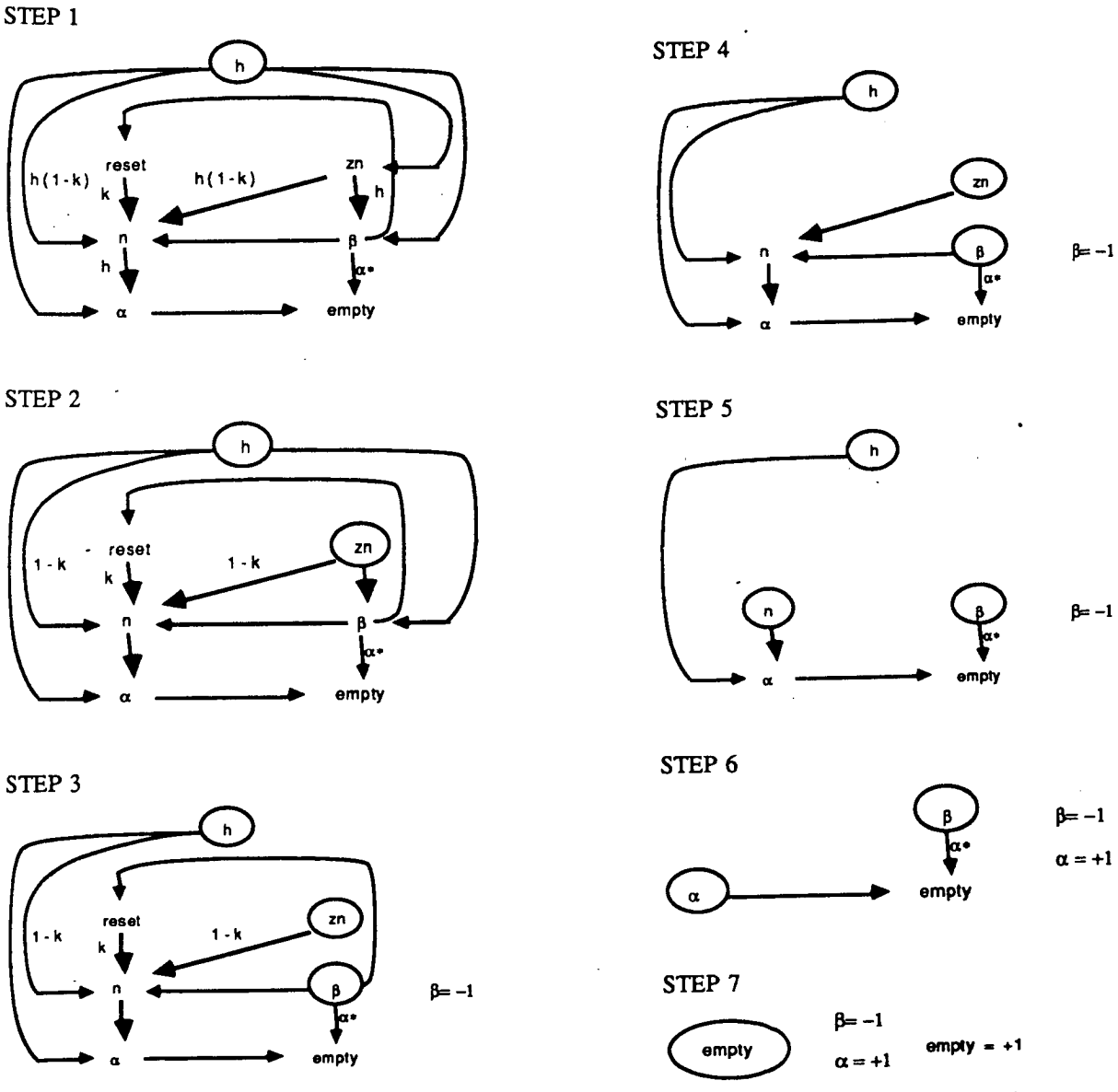


Figure 6: running EXEC on the program GUARDED_COUNT

4.4 Handling deadlocks: a fundamental theorem.

A natural question arises about the preceding algorithm, namely *does this algorithm terminate correctly, i.e. with all nodes and clocks being evaluated or absent?* This is the question we want to answer in this section.

4.4.1 The clock of a cycle of G.

Consider a node x in the graph G of the considered synchro-process, and assume it satisfies the following property:

$$\{P : y \longrightarrow x\} \in CDG \text{ and } x \text{ boolean,}$$
$$\text{referred to as } \text{«x free boolean»} \tag{4-10}$$

In other words, x is a boolean which is the result of the evaluation of a non boolean function (for instance x := (u<v)), hence the name of «free» since our clock calculus cannot compute its actual value. The clock of a cycle of G is defined now.

STEP 1: Consider a cycle

$$C = X_1 \xrightarrow{P_1'} X_2 \ldots X_n \xrightarrow{P_n'} X_1$$

of G, and denote by Ξ the (possibly empty) subset of the X_i's which are *free booleans*.

STEP 2: Each equation $P_i' = 0$ defines an algebraic variety V_i'. Denote by V_i the smallest variety containing V_i' which is *invariant* by the group of symmetries $X \to -X \ \forall X \in \Xi$, and denote by P_i the polynomial such that $P_i = 0$ defines the variety V_i.

STEP 3: the clock of the considered cycle is defined as

$$\textit{clock}(C) = \prod_{i=1}^{n} P_i \tag{4-11}$$

In other words, inside a cycle of G, we extend the dependencies to the least frequent clock which is
1/ more frequent than the product of the original clocks of the branches
2/ independent of the actual values of the free booleans which will be evaluated within the considered cycle.
Obviously, if no free boolean belongs to the nodes of this cycle, we just get the original product.

IMPORTANT REMARK: since the graph of a synchro-process is not invariant via isomorphisms, the cycles depend on the particular form of the clock calculus. Hence it is expected that some cycles can be broken[2] via suitable manipulations of the clock calculus. This idea will be exploited further.

4.4.2 A fundamental theorem to analyse deadlocks.

Fundamental theorem 4: *the conditions (i,ii,iii) below ensure that EXEC terminates correctly for the process P, i.e. that all nodes and clocks have either been evaluated or proved to be absent within the considered instant:*

(i): the clock calculus is pre-solved;

(ii): P is observable with respect to the observer composed by its source nodes and free booleans;

(iii): all cycles of G have zero clock.

In other words, processes which are observable by their inputs as well as cycle free in the sense of (iii) are deadlock free. The proof is given in the appendix.

4.5 Solving clock calculi: the algorithm COMPIL.

Our purpose here is to investigate how to transform the clock calculus of any synchro-process to get the form mentioned in the fundamental theorem, i.e. suitable to a correct termination of EXEC. To help for the resolution, we need to handle graphs for synchro-processes which are not in the pre-solved form: we shall use the rule GRAPH_4 for this purpose.

Then the clock calculus is solved in the following way.

STEP 1: perform all possible substitutions of the left handside by the corresponding expression in the righthandside of $YorY^2 = \ldots$ until implicit equations are encountered. When several equations of the form $YorY^2 = \ldots$ are encountered, select one as the definition equation of the left handside and form a constraint by expressing that the two righthandsides must be equal (an elementary way of performing elimination). Definition equations of the form $YorY^2 = P(freebool)$ where P is any polynomial and *freebool* denotes any free boolean are preferred in the case of selecting a definition equation among several ones.

STEP 2: Build the graph G of the so obtained synchro-process. Define on the set of the vertices of G the following equivalence relation denoted by $x \leftrightarrow y$: $x \leftrightarrow x$ and $x \leftrightarrow y$ if x and y belong to the same strong connectivity class (i.e. there is a path from x to y and vice-versa). Then G/$\leftrightarrow$ is

---

2 see the definition of the actions of EXEC

a circuit-free graph; denote by $\{G_i\}_{1 \le i \le n}$ the subgraphs of G which are mapped onto vertices of G/↔ where the index n is compatible with the partial order on these subgraphs. These subgraphs will be simply called *strong connectivity classes* in the sequel. Denote by

$$C_{k_1}, \ldots, C_{k_p}$$

the circuits of the G_i's which possess at least one branch of one of the forms

$$X^2 \xrightarrow{P} \text{label} \quad (\text{originating from GRAPH\_1})$$
$$\text{label} \xrightarrow{P} \text{label} \quad (\text{originating from GRAPH\_2})$$

Such cycles will be called *data-cycles*: they cannot be broken[3] by transformations of the clock calculus. Hence for each data cycle, we must add the following constraint which ensures the condition (iii) of the main theorem:

$$C \text{ data cycle} \Rightarrow \text{add } \mathit{clock}(C) = 0 \text{ to the clock calculus} \qquad (4\text{–}12)$$

where the clock of C has been defined in (4–11). Notice that (4–12) generally modifies the graph G, *but does not add new data-cycles*. After STEP 2, data-cycles are broken.

STEP 3: Using the rules for elimination of lemma 5, the remaining connectivity classes are broken successively, starting from the last one (according to the partial order induced by G). This is done as follows. Data-cycles are not modified since they have already been handled. Elimination within a connectivity class terminates with the variables of the class which are successors of nodes of G which do not belong to the class (the «source nodes of the connectivity class»).

RESULT: if this procedure terminates with no phantom, the assumptions of the fundamental theorem are satisfied.

The procedure we have presented informally is discribed in the appendix via the technique of transition systems. This procedure will be called COMPIL in the sequel.

DISCUSSION: as it will be shown in the examples, this procedures isolates the subset of the ports that are deadlocked in the considered process. Hence this procedure is a fundamental tool for programming fault isolation. The resulting graph is also a convenient starting point to target the considered application on a multiprocessor architecture, see [Figueira & al. 1988].

---

3 see the definition of the actions of EXEC

4.6 Examples.

Our purpose in this section is to show how the preceding procedure handles spurious programs, to detect and isolate deadlocks, or transform a program into an executable form. Hence some pathological examples will be reviewed.

4.6.1 A wrong synchronization.

Recall the following example which has been introduced in [Benveniste & Le Guernic * 1988]:

```
x := u when (u<v)
| y := x + v
```

Writing β for short instead of (u<v), we get as clock calculus

$$y^2 = x^2 = u^2 = v^2 = \beta^2 = h$$
$$x^2 = u^2(-\beta - \beta^2)$$

Due to this clock calculus, the graph of this program exhibits a cycle, namely

$$h \xrightarrow{-\beta-\beta^2} h$$

Hence the clock of this cycle has to be zero. But this clock is equal to

$$(-\beta-\beta^2)^2 + (\beta-\beta^2)^2 = \beta^2$$

Which yields $\beta = 0$, hence the process stays in deadlock. In this example the synchronisation was errored.

4.6.2 A data-cycle.

The following example is due to G. Gonthier (private communication); it roughly means

```
if z>0 then z:=a else z:=b
```

This program should be rejected. A suitable SIGNAL program is

```
synchro a,b
| β = (z>0)
| x := a when β
```

y := b *when not* β
z := x *default* y

The clock calculus is

$$a^2 = b^2 = h$$
$$x^2 = h(-\beta - \beta^2)$$
$$y^2 = h(\beta - \beta^2)$$
$$\beta^2 = z^2 = x^2 + y^2(1 - x^2) = h\beta^2$$

The graph is

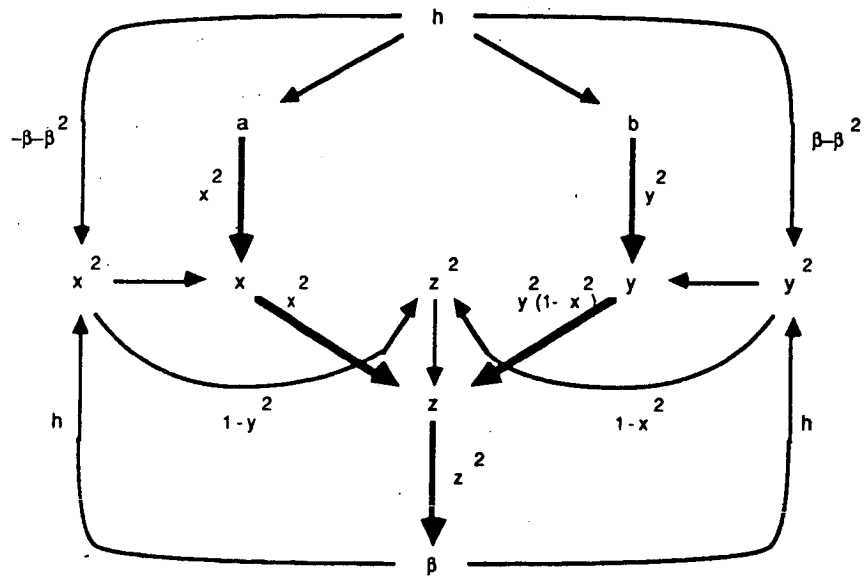


Figure 7: the graph of Gonthier's example.

Two cycles are exhibited; their clocks are both equal to $h\beta^2$. Hence these cycles add to the clock calculus the condition $\beta = 0$. As a result, this program accepts the inputs a,b, but refuses to produce any other signal. The isolated deadlock involves the signals x,y,z,β; such an isolation is a help for a fault isolation.

4.6.3 A case of data-cycle without deadlock.

Consider the following (spurious) program

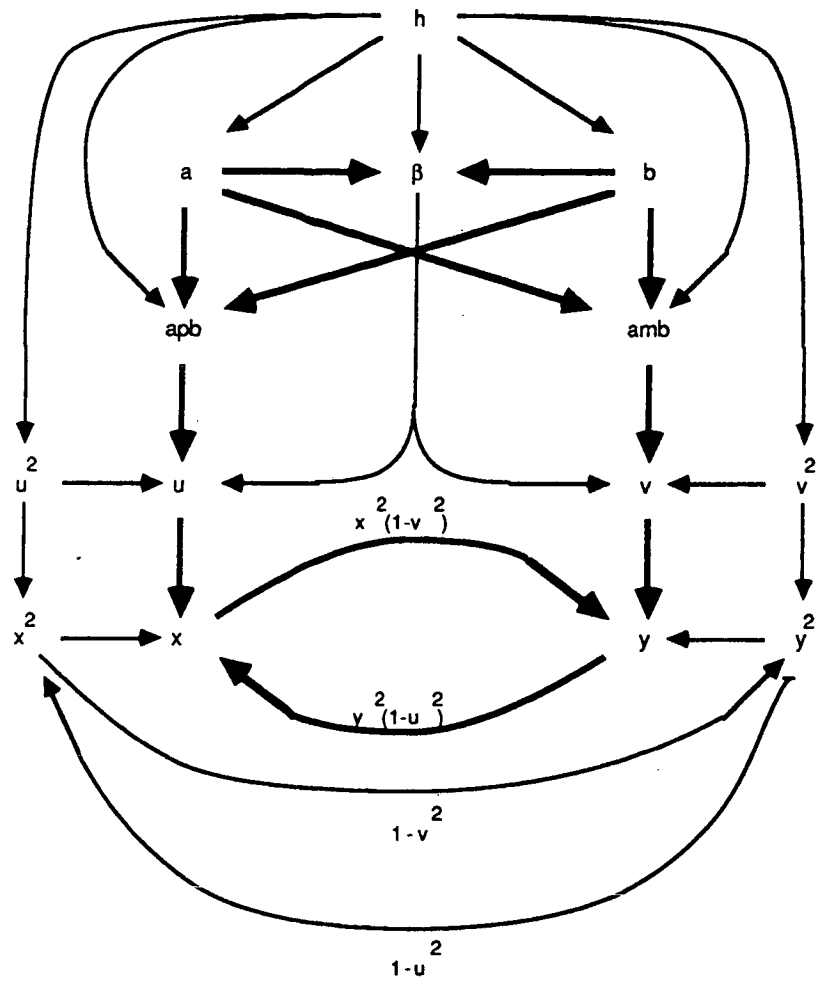
β := (a<b)

```
apb := a+b
u := apb when β
x := u default y
amb := a-b
v := amb when not β
y := v default x
```

The clock calculus is

$$a^2 = b^2 = \beta^2 = apb^2 = amb^2 = h \quad (i)$$
$$u^2 = h(-\beta - \beta^2) \quad (ii)$$
$$v^2 = h(\beta - \beta^2) \quad (iii)$$
$$x^2 = u^2 + y^2(1 - u^2) \quad (iv)$$
$$y^2 = v^2 + x^2(1 - v^2) \quad (v)$$

The graph is



NOTA: only the clocks on the branches of cycles have been shown.

Figure 8: the graph of a spurious program.

This graph exhibits two cycles; one of them is a cycle due to the clock calculus $x^2 \rightarrow y^2 \rightarrow x^2$ and will be handled by the elimination; the other one $x \rightarrow y$ is a data-cycle which enforces the condition

$$y^2 x^2 (1 - u^2)(1 - v^2) = 0 \quad (vi)$$

Elimination is first performed with the subset of equations (iv,v,vi) which corresponds to a strong connectivity class of the graph; the elimination is performed by considering u, v as free variables since they precede this class. The rules of COMPIL_ELIM give finally

$$x^2 = u^2 + (1 - v^2)$$
$$y^2 = v^2 + (1 - u^2)$$

which removes the cycle due to clock equations. If desired, further simplification using COMPIL_REDUCE yields the reduced clock calculus

$$a^2 = b^2 = \beta^2 = apb^2 = amb^2 = x^2 = y^2 = h \quad (i)$$
$$u^2 = h(-\beta - \beta^2) \quad (ii)$$
$$v^2 = h(\beta - \beta^2) \quad (iii)$$

The algorithm EXEC can run on this process. Notice that the ultimate form of this spurious program is exactly

if a<b *then* y:=a+b *else* y:=a–b

which would have been directly proposed by any healthly programmer!

Chapter Five
Conclusion.

We have presented the models supporting our HDS theory. The modelling via transition systems has been shown very close to an algebra of distributed and communicating automata. On the other hand, the coding via transition systems has been repeatedly used to specify algorithms acting as state machines, thus illustrating to effectiveness of this approach. The model Ω emphazes on relations between trajectories of signal flows and handles these relations. This second model provides a natural introduction to our algebraic coding of programs as dynamical systems over F_3. The core of the theory is HDS resolution which converts a (non executable) dynamical system specification into an executable machine; HDS resolution has been shown to cover the notion of exact model following control.

Further work to be pursued concerns the following points:

- Exploit and study more deeply the temporal properties of the clock calculus. It is desirable to weaken the notion of observability we have introduced, since the latter is a strictly static notion. Phantoms have been introduced as a criterion to recognize observability in our sense; but phantoms can be used as well to locate the possible places where *control can be applied* consistently with the considered HDS to reduce the non determinacy of the system (this is a point of view similar to Ramadge and Wonham's disabling/enabling technique, although the supervisors that are construct in our case are always controllable).
- Study how our dynamical clock calculus can cover notions from temporal logic [Pnueli 1977]; this would provide SIGNAL as a tool for HDS synthesis from specifications in a style similar to temporal logic. In fact the «shared track» example of [Benveniste & Le Guernic * 1988] showed some flavour of such a synthesis.
- Other fundamental homomorphisms could be of interest than encoding non boolean functions via dependence graphs. A possibility would be to encode numerical calculi into the physical duration they need to be performed. By the way a simulator is obtained to verify the timing constraints. But if this coding uses the algebraic coding via *dioids* as introduced by [Cohen et al. 1986] we obtain a kind of skew product of clock calculus and dioids we hope to be able to study in the future.
- Finally the Ω model is obviously ready to accept probabilities, thus providing a basis for stochastic HDS.

To conclude, let us mention the works around other *synchronous* languages, i.e. languages which rely deeply on the notion of «instant» to perform complex macro−actions requiring the solution of some systems of symbolic equations.

Among synchronous languages is the other declarative language LUSTRE [Bergerand & al. 1985] [Caspi et al. 1987], originating weakly from the functional language LUCID to handle sequences [Ashcroft & Wadge 1976]. The major difference with SIGNAL lies in the *functional* nature of LUSTRE compared to the relational one of SIGNAL. A consequence is that our clock calculus is replaced in LUSTRE by clock verification since no LUSTRE subprocess can be nondeterministic, and HDS synthesis in the sense we have discussed cannot be performed. Nevertheless dependencies have to be synthetized as in our case. As a result LUSTRE requires a simpler compiler than SIGNAL, and is still convenient to program numerous real time tasks relevant to signal processing or control.

On the other hand the imperative language ESTEREL [Berry & Cosserat 1984][Gonthier 1988] possesses a syntax with some flavour of ADA. As an imperative language, ESTEREL is highly suited to a coding via transition systems and uses a fast implementation of Kleene's realization algorithm for regular languages to be compiled into an automaton performing actions which can be numerical computations. ESTEREL is *reactive* in the sense that an ESTEREL program cannot constrain the clocks nor the (boolean) values of its input stimuli. Hence this language cannot be used for HDS synthesis in our sense, but is highly efficient to build complex systems of interconnected automata performing symbolical or numerical actions. Deadlocks as well as nondeterminism (unobservability in our sense) are checked as well.

ACKNOWLEDGEMENT: the authors are indebted to the SIGNAL working group for valuable discussions and permanent exchanges of ideas; they wish to thank M. Sorine and Y. Sorel from INRIA-Rocquencourt, T. Gautier and B. Cheron from IRISA, and especially A.S. Willsky and C. Ozveren from MIT who are currently visiting IRISA.

Chapter Six
Appendix.

6.1 Proof of the fundamental theorem.

The proof proceeds along a set of lemmas. Conditions (i,ii,iii) are assumed to be in force.

Lemma 6: EXEC never terminates in the following state:

$$\exists Y \in nodeG \text{ such that } Y \in ?(nodeG) \text{ and}$$
$$\forall \{X \xrightarrow{P} Y\} \in G:\ X \in val(nodeG), P \in val(CLOCK) \qquad (6\text{-}1)$$

Proof: the lemma is immediate if Y is the result of a non-boolean function, since it is just rule EXEC_1. On the other hand if Y is to be evaluated by the clock calculus, the considered branch must be caused by rule GRAPH_3 used to construct G. Then (4-8) and (6-1) implies that all variables on the right handside of the definition of Y or Y^2 have been evaluated, which means that Y or Y^2 must have been already evaluated using one of the rules EXEC_0 or EXEC_1. This proves the lemma.

Lemma 7: Consider the following statement $\mathbf{P}_1$:

$$Y \in ?(nodeG) \text{ and } \exists X \in val(nodeG) : \{X \xrightarrow{P} Y\} \in G$$
$$\Downarrow$$
$$\exists Z \in ?(nodeG) : \{Z \xrightarrow{Q} Y\} \in G$$

Then, if *not* $\mathbf{P}_1$ holds at every step of EXEC, this algorithm terminates correctly.

Proof: It suffices to prove that the following case cannot occur: $\exists Y$ such that

$$\forall \{X \xrightarrow{P} Y\} \in G : X \in val(nodeG)$$
$$and$$
$$\exists \{X \xrightarrow{Q} Y\} : Q \in ?(CLOCK) \qquad (6\text{-}2)$$

CASE 1: Y is a label; then the above mentioned branches have been obtained using rule GRAPH_2 which implies that all clocks labelling the incoming branches belong to the set of the predecessor nodes of Y so that they are evaluated.
CASE 2: Y is a variable of the clock calculus. Then by rule GRAPH_3 and the first statement of (6-2) all variables on the right handside of the definition of Y must have been evaluated which implies that Y also must have been evaluated.

To state the next lemma, we organize G into strong connectivity classes (sets of nodes which are successors of each other); the set of the strong connectivity classes is denoted by $\mathcal{C}$ and elements of $\mathcal{C}$ are generically denoted by C. We write $val(nodeG) \longrightarrow C$ to mention that some of the nodes of C are successors in G of the subset of evaluated nodes. Finally, «minimal» has to be taken in the sense of the partial order on strong connectivity classes.

Lemma 8: $P_1 \Rightarrow P_2$ where P_2 is the property

$$C \in \mathcal{C} \text{ minimal such that } val(nodeG) \longrightarrow C$$
$$\Downarrow$$
$$C \subset ?(nodeG)$$

Proof: Immediate.

In the next lemma, we consider a strong connectivity class which satisfies P_2 at the considered step of EXEC, and we denote it by C. A branch of C is said to be *broken* if its clock label has already been evaluated as 0.

Lemma 9: The set of broken branches in C is observable by the subset of the elements of *val*(G).

Proof: Thanks to condition (ii) of the theorem, the clock labels of the branches of C are known as soon as

- free variables of the clock calculus
- actual values of the free booleans

are known. Moreover thanks to the rules that have been used to build G, and using the minimality of C, it is sufficient to know those of them which are predecessors or members of C. But, the clock labels of the branches of any cycle of C do not depend on the actual values of the free booleans to be evaluated within C (cf the definition of the clock of cycles). Hence the lemma.

End of the proof of the theorem: By assumption (iii) of the theorem, we know that all cycles of C must have zero clock, i.e. can be broken for all admissible set of values for the clock labels. In the following picture, only the considered strong connectivity class is depicted; the incoming branches are thus originating from nodes that have been evaluated.

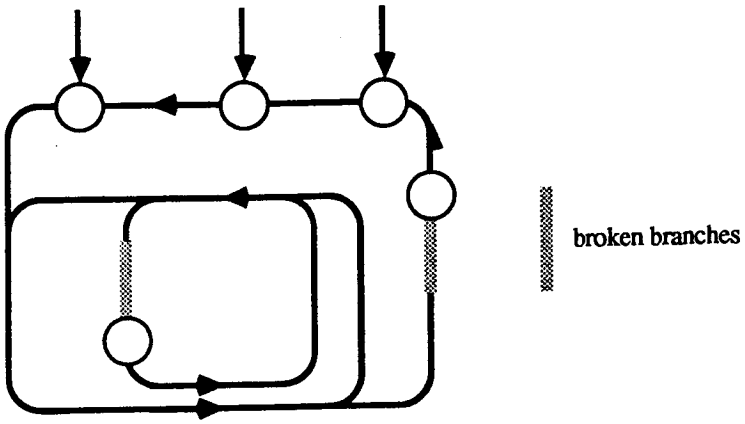


Figure 9: a prohibited situation, broken branches in the interior only.

We know by lemma 9 that broken branches of C must be known before hitting C. But lemma 6 prevents cycles to be broken in the interior of C only, see the picture above. Hence we know before to hit C that it will be broken at its boundary and we know where this will occur, see the following picture:

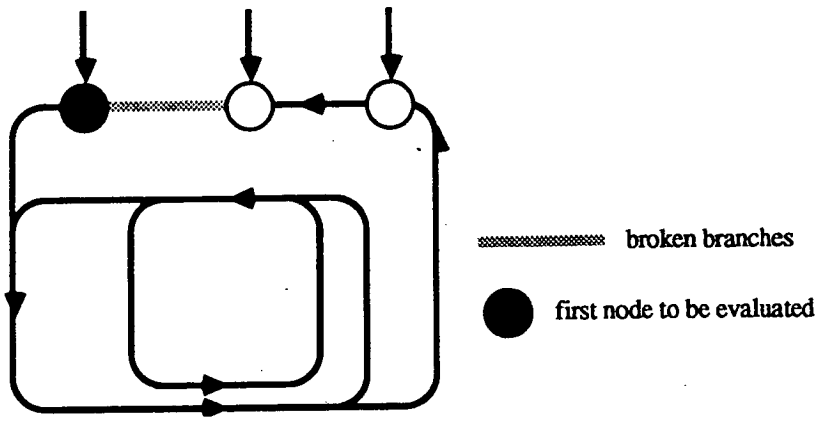
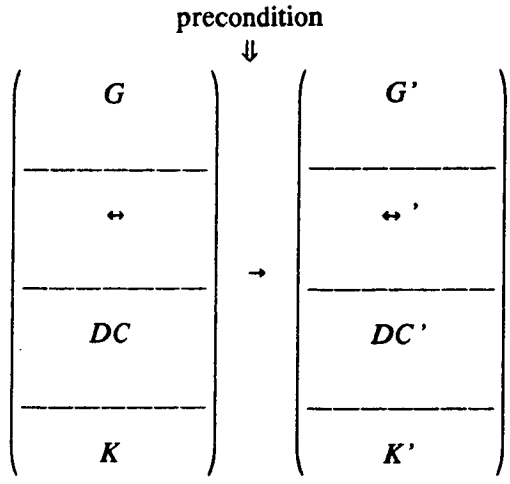


Figure 10

Hence an entry point of C actually does exist, which contradicts P_2 hence P_1. This finally proves the theorem.

6.2 Presentation of macro–rules for COMPIL.

These macro-rules will be presented using transition systems of the form:



where the fields of the states are

1. G: the current graph.
2. $\leftrightarrow$: the relation defined in the informal description of COMPIL.
3. DC: the set of data-cycles together with their elements; it should be noticed that $DC \neq \emptyset$ should correspond to errored or at least spurious programs; hence the relatively high complexity of constructing DC when it is non empty can be accepted.
4. K: the current state of the clock calculus.

6.2.1 Initial and final states.

Initial state.

$$\left(\begin{array}{c} G = G(PROG) \\ \hline \leftrightarrow = \leftrightarrow(G) \\ \hline DC = DC(G) \\ \hline K = K(PROG) \end{array}\right)$$

G(PROG) is the graph of the program as obtained via the rules GRAPH_1 to _4; *K(PROG)* is the clock calculus of the program as obtained via the algebraic coding of its instructions.

Final state. The pair G_{end}, K_{end} is the synchro-process to which the algorithm EXEC will be applied: the conditions (i) and (iii) of the fundamental theorem are satisfied, and it can be immediately checked whether condition (ii) holds (check for the presence of phantoms in K_{end}).

$$\left(\begin{array}{c} G_{end} \\ \hline \leftrightarrow = \{x,x\} \forall x \in nodeG \\ \hline DC = \emptyset \\ \hline K_{end} \end{array}\right)$$

6.2.2 Rule COMPIL_DC for handling data-cyles.

$$cycle \in DC \Downarrow$$

$$\left(\begin{array}{c} G \\ \hline \leftrightarrow \\ \hline DC \\ \hline K \end{array}\right) \rightarrow \left(\begin{array}{c} G \\ \hline \leftrightarrow \\ \hline DC - cycle \\ \hline K \cup \{clock(cycle) = 0\} \end{array}\right)$$

This rule is first repeatedly used until all data–cyles are removed. Hence, for the subsequent rules, the field *DC* can be removed.

6.2.3 Rules COMPIL_MULT_ for handling multiple definitions.

RULE COMPIL_MULT_1

$$\{h: x \longrightarrow y\} \in CDG \text{ and } \{y = \bullet \in K\} \Downarrow$$

$$\left(\begin{array}{c} G \\ \hline \leftrightarrow \\ \hline K \end{array}\right) \rightarrow \left(\begin{array}{c} G \cup \{y \longrightarrow any \bullet\} \\ \hline \leftrightarrow \cup \{y, any \bullet\} \\ \hline K \end{array}\right)$$

This rule is concerned with the case of free boolean which appears also on the left handside of some definition equation $y = \bullet$ of the clock calculus. In this case, the considered node y influences any variable of $\bullet$, which causes this set of variables together with y to be added to some connectivity class.

RULE COMPIL_MULT_2

$$\{y = \bullet_1 \in K\} \text{ and } \{y = \bullet_2 \in K\}$$

$$\Downarrow$$

$$\left(\begin{array}{c} G \\ \hline \leftrightarrow \\ \hline K \end{array}\right) \rightarrow \left(\begin{array}{c} G \cup \{y \longrightarrow any \bullet_1, y \longrightarrow any \bullet_2\} \\ \hline \leftrightarrow \cup \{y, any \bullet_1, any \bullet_2\} \\ \hline K \end{array}\right)$$

This rule handles the case of variables of the clock calculus which appear on the left handside of two different equations. The proposed modification of the two fields G,C takes into account the constraint caused by this double definition. After repeated use of rules COMPIL_DC and COMPIL_MULT_, we are ready to proceed with elimination, as described by the forthcoming rule.

6.2.4 Performing elimination by rule COMPIL_ELIM.

In the following, *class* denotes a strong connectivity class, and *last* ($\leftrightarrow$) denotes the set of the classes which are the last ones in the partial order of the connectivity classes; *pairs* (*class*) denotes the set of the pairs of nodes of *class*. Finally, *pairs* (***elim_constraint*** (*class*)) will be defined later.

$$class \in last\ (\leftrightarrow)$$

$$\Downarrow$$

$$\left(\begin{array}{c} G \\ \hline \leftrightarrow \\ \hline K \end{array}\right) \rightarrow \left(\begin{array}{c} G'(G, elim) \\ \hline (\leftrightarrow - pairs\ (class)) \cup pairs\ (\textbf{\textit{elim\_constraint}}\ (class)) \\ \hline K'(K, elim) \end{array}\right)$$

Hence the elimination is performed by beginning with the classes which are far from the inputs and ending by the input classes. In the resulting state,

- *G'(G,elim)* is the graph obtained after all the manipulations of K caused by this rule: just apply rules GRAPH_3 and _4 to add the new branches.
- ***elim_constraint(class)*** is the set of clock equations caused by the elimination performed in *class*, see the details for this rule here after.

- *pairs(elim_constraint(class))* is the modification of the connectivity classes caused by this rule; while *class* has been definitely removed from $\leftrightarrow$, the latter modification concerns only the classes that are predecessors of *class*.

Details on this rule will be provided later. After repeated uses of the rule COMPIL_ELIM, we end up with a synchro-process satisfying conditions (i) and (iii) of the fundamental theorem. Whenever desired, the following last rule provides us with *the minimal form* of the considered process which is suitable to EXEC. Since elimination has been performed, the field C is no more useful, and can be deleted.

6.2.5 Reducing the synchro-process by rule COMPIL_REDUCE.

$$\left(\begin{array}{c} G \\ \hline K \end{array}\right) \rightarrow \left(\begin{array}{c} G_{end} \\ \hline K_{end} \end{array}\right)$$

The purpose of this rule is to rewrite K in terms of free variables of the clock calculus only, i.e. input clocks or booleans, as well as free booleans that are evaluated while running EXEC. The graph G is modified accordingly. This final step will allow ultimately to remove some of the phantoms that might have been produced during the elimination. Hence condition (ii) of the fundamental theorem can be more properly checked after this reduction step. This rule is performed via standard rewriting.

6.3 Details for COMPIL_ELIM.

Since the manipulations on K are the cause of all changes in the other fields G,C, we only present the modifications performed on K. Denote by *EQ(class)* the subset of the clock equations which caused *class* to occur, and denote by Ξ the set of the variables involved in *EQ(class)*.

6.3.1 Ordering the equations of *class*.

Write *cl* for short instead of *class*. In *cl* we distinguish

1. *in(cl)*: the subset of the nodes of *cl* which are immediate successors of other connectivity classes;
2. *out(cl)*: the subset of the nodes of *cl* which do not belong to *in(cl)* and are immediate predecessors of other connectivity classes.

Then select any ordering on Ξ which is compatible with the decomposition above: select at first the elements of *out(cl)* and at last the elements of *in(cl)*.

Ordering *EQ(class)*.
Given this ordering, an ordering on *EQ(class)* is defined as follows.

STEP 1: order on the monomials.
Monomials M are partitionned according to their *head* and their *tail*: $M = \{X or X^2\}.N$ where the tail N is an other monomial.

Order on the heads: $X_1 < X_1^2 < X_2 < X_2^2 < \ldots$ except that *any* X_i *with degree 1 which is a free boolean has to be rejected at the end.*

Order on the tails: the tail is ordered in reverse lexicographic order with increasing degrees, i.e. (here $i_k = 0, 1, 2$), using tryadic expansions,

$$\prod_{k=1}^{K} X_k^{i_k} < \prod_{k=1}^{K} X_k^{j_k}$$
$$\Leftrightarrow$$
$$\sum_{k=1}^{K} i_k 3^{K-k} < \sum_{k=1}^{K} j_k 3^{K-k}$$

STEP 2: order on the equations. Equations are ordered via the comparison of their first monomials (i.e. the least with respect to $.<.$).

REMARK: while the ordering of the variables is of importance since this ordering will fix the order of the elimination, the ordering of the equations is selected for the convenience: its purpose is to force the «simplest» equations to be considered first when several ones involve the same variable to be eliminated.

6.3.2 Initial and final state.

Initial state: *K(cl)* denotes the set of the clock equations where elements of *cl* are involved, and the empty field is intended to contain the equivalent system in triangular form.

$$\left(\begin{array}{c} K(cl) \\ \hline \varnothing \end{array}\right)$$

Final state: the first field contains the constraints on the variables which are predecessors of *cl*, that are caused by the elimination. These constraints are mentioned in the rule COMPIL_ELIM with the name of ***elim_constraint(class)***. The second field contains the solved calculus, i.e. the equivalent triangular form. The fields "G" and "C" are modified accordingly to get the new mentioned fields respectively denoted by *G'(G,elim)* and ***C(elim_constraint(class))***.

$$\left(\begin{array}{c} elim\_constraint\,(class) \\ \hline solved\_K(cl) \end{array}\right)$$

6.3.3 The rules.

Ordering the equations.

$$\left(\begin{array}{c} K \\ \hline s\_K \end{array}\right) \rightarrow \left(\begin{array}{c} first\_K; tail\_K \\ \hline s\_K \end{array}\right)$$

Rules for simple substitutions.

$$\left(\begin{array}{c} X = \bullet; K \\ \hline s\_K \end{array}\right) \rightarrow \left(\begin{array}{c} K(X \leftarrow \bullet) \\ \hline s\_K; X = \bullet \end{array}\right)$$

The notation $K(X \leftarrow \bullet)$ means that X is substituted by its expression $\bullet$ in K. This notation will be used repeatedly in the sequel.

$$\left(\begin{array}{c} X^2 = \bullet; K \\ \hline s\_K \end{array}\right) \rightarrow \left(\begin{array}{c} K(X^2 \leftarrow \bullet) \\ \hline s\_K; X^2 = \bullet \end{array}\right)$$

Notice that this second rule is used only when the first one has not be used: if $X = \ldots$ and $X^2 = \ldots$ are both present, then the first one is processed first thanks to rule for rewriting X, so that X disappears from the remaining equations.

Fusion of equations.

$$X \text{ not free boolean}$$
$$\Downarrow$$
$$\begin{pmatrix} aX^2+bX+c=0 \\ a'X^2+b'X+c'=0 \\ K \\ \hline s\_K \end{pmatrix} \rightarrow \begin{pmatrix} (aX^2+bX+c)^2+(a'X^2+b'X+c')^2=0 \\ K \\ \hline s\_K \end{pmatrix}$$

Solving for clocks.

$$X \text{ first variable in } K$$
$$\Downarrow$$
$$\begin{pmatrix} aX^2+c=0;K \\ \hline s\_K \end{pmatrix} \rightarrow \begin{pmatrix} c(a+c)=0;K \\ \hline s\_K;aX^2+c=0 \end{pmatrix}$$

We choosed to keep the original form of the equation as the definition of X^2, but it should be clear that the lemma 5 gives an explicit definition if desired. Such an explicit form should be preferred if one whishes to run EXEC, but the present implicit one is useful when our purpose is to perform «separate compilation», i.e. to compile separately different processes before connecting the so-compiled subprocesses.

Solving for non free booleans.

$$X \text{ non free boolean, and first variable in } K$$
$$\Downarrow$$
$$\begin{pmatrix} aX^2+bX+c=0;K \\ \hline s\_K \end{pmatrix} \rightarrow \begin{pmatrix} c[(a+c)^2-b^2]=0;K \\ \hline s\_K;aX^2+bX+c=0 \end{pmatrix}$$

The same remark holds as before.

6.4 Discussion.

The different rules of COMPIL, as well as the construction of K and G directly from the program can be combined and pipelined to avoid unnecessary duplications of tasks. For example the construction of ↔, DC can be performed while constructing G itself. The strict

down → *up* → *down*[4] procedure suggested by the successive use of COMPIL_ELIM and then COMPIL_REDUCE can be replaced by some interleaving of these. Such an optimisation of the compilation algorithm requires further work.

REFERENCES.

[Ashcroft & Wadge 1976]: E.A. Ashcroft, W.W. Wadge, «LUCID – a formal system for writing and proving programs», SIAM J. Comp., vol 5 No 3, 336–354.

[Benveniste & Le Guernic 1987]: A. Benveniste, P. Le Guernic, «A denotational theory of synchronous communicating systems», INRIA research report No 685.

[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*.

[Bloom & al. 1987]: B. Bloom, S. Istrail, A.R. Meyer, «Bisimulation can't be traced», Proceedings of POPL'88.

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

[Boudol & Castellani 1986]: G. Boudol, I. Castellani, «On the semantics of concurrency: partial orders and transition systems», INRIA res. rep. No 550.

[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.

[Buchberger 1970]: B. Buchberger, «Ein algorithmisches Kriterium fur die Losbarkeit eines algebraisches Gleichungssystems, Aequat. Math. 4, 374–383.

[Buchberger 1979]: B. Buchberger, «A criterion for detecting unnecessary reductions of Grobner bases», Eurosam 79, Lect. Notes in Comp. Sc. vol 72, 3–21, Springer Verlag.

[Caspi et al. 1987]: P. Caspi, D. Pilaud, N. Halbwachs, J.A. Plaice, «LUSTRE: a declarative

---

[4] i.e. beginning the elimination from output to input ports, and then performing the reduction from input to output ports

language for programming synchronous systems», Proc of the 14th ACM symp. on Principles of Programming Languages, 1987.

[Caspi & Halbwachs 1986]: P. Caspi, N. Halbwachs, «A functional model for describing and reasoning about time behaviour of computing systems», Acta Informatica 22, 595-627.

[Cohen et al. 1986]: G. Cohen, P. Moller, J.P. Quadrat, M. Viot, «Dating and couting in discrete event systems», 25th CDC, Athens.

[Dellacherie & Meyer 1976]: C. Dellacherie, P.A. Meyer, *«Probabilites et Potentiel»*, 2nd edition, Hermann, Paris.

[Figueira & al. 1988]: C. Figueira, T. Gautier, B. Le Goff, P. Le Guernic, "Towards multiprocessor implementation of real-time, data-flow programs", The 1988 International Symposium on LUCID and Intentional Programming, Victoria, Canada, April 6,7,8, 1988.

[Gautier 1987]: T. Gautier, P. Le Guernic, L. Besnard, «SIGNAL, a Declarative Language for Synchronous Programming of Real-time Systems», proc. of the third Conference on Functional Programming Languages and Computer Architecture, G. Kahn Ed, Lect Notes in Computer Science, vol 274, Springer V.

[Gonthier 1988]: G. Gonthier, thesis, Univ. de Nice and Ecole des Mines, 1988.

[Kahn 1974]: G. Kahn, «The semantics of a Simple Language for Parallel 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.

[Le Guernic & Benveniste 1986]: P. Le Guernic, A. Benveniste, «Real-time synchronous data-flow programming: the language SIGNAL and its mathematical semantics», INRIA research report No 533, reprint as INRIA res. rep. No 620.

[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.

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

[Milner 1983]: R. Milner «Calculi for Synchrony and Asynchrony», Theoretical Computer Science, 25 No 3, 267-310.

[Ornstein 1974]: D.S. Ornstein *Ergodic Theory, Randomness, and Dynamical Systems*, Yale Univ. Press, 1974.

[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.

[Ramadge 1986]: P.J. Ramadge, «Observability of discrete event systems», CDC-1986, Athens, 1108-1112.

[Ramadge and Wonham 1987-a]: P.J. Ramadge, W.M. Wonham, «Supervisory Control of a class of Discrete Event Processes», SIAM J. Control and Opt., 25, 1, 1987, 206-230.

[Ramadge and Wonham 1987-b]: P.J. Ramadge, W.M. Wonham, «On the Supremal Controllable Sublanguage of a Given Language», SIAM J. Control and Opt., 25, 3, 1987, 637-659.

[Willems 1987]: J.C. Willems «From time series to linear systems», Automatica

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