Programming Real-Time Applications with Signal

Paul Le Guernic, Thierry Gautier, Michel Le Borgne, Claude Le Maire

Programming real time applications with SIGNAL

Paul Le Guernic Thierry Gautier
Michel Le Borgne Claude Le Maire

IRISA
Campus de Beaulieu
35042 Rennes cedex
FRANCE

Abstract

This paper presents the main features of the Signal language and its compiler. Designed to provide safe real time system programming, the Signal language is based on the synchronous principles. Its semantics is defined via a mathematical model of multiple-clocked flows of data and events. Signal programs describe relations on such objects, so that it is possible to program a real time application via constraints. The compiler calculates the solutions of the system and may thus be used as a proof system. Moreover, the equational approach is a natural way to derive multiprocessor executions of a program. Finally, this approach meets the intuition through a graphical interface of block-diagram style, and the system is illustrated on a speech recognition application.

1 Introduction

Signal is a block-diagram oriented synchronous language for real time programming. According to the *synchronous* approach, time is handled according to the first two of its three following characteristic aspects: partial order of events, simultaneity of events, and finally delays between events. In a synchronous framework, time is modeled as a *chronology;* durations are constraints to be verified at the implementation. Then it is possible to consider that computations (and in particular computations about time) have zero duration. This hypothesis is acceptable if any operation of ideal zero duration has a bounded effective duration. We refer the reader to [1] for a discussion of the principles of synchronous programming. As discussed in this introductory paper, the styles of synchronous languages may be classified into *imperative* ones and *equational* ones. The first style relies on models of the state-transition machine family. CSML [17], Esterel [18], and the Statecharts [20] follow this style. The second one relies on models of multiple-clocked interconnected

dynamical systems. LUSTRE [19] follows this style, based on a strictly *functional* point of view. In SIGNAL, programming is performed via the specification of constraints or *relations* on the involved signals. As a consequence, the SIGNAL compiler performs *formal calculations on synchronization, logic, and data dependencies* to check program correctness and produce executable code.

The paper is organized as follows. The section 2 is devoted to an informal presentation of the main features of the language. The mathematical model supporting SIGNAL is briefly discussed in section 3, further information may be found in [2, 3, 4, 11]; based on this formal model, it is explained how the SIGNAL compiler operates. Distributed code generation is discussed in the section 4. Finally, a speech recognition application that was introduced in [1] is described in section 5.

2 The language

In this section we introduce the reader to programming in SIGNAL. For that purpose, we investigate the two examples introduced in [1], namely the *digital filter* and the *mouse*. Finally the use of SIGNAL as a proof system to verify temporal properties is introduced in the last subsection.

The SIGNAL language handles (possibly infinite) sequences of data with time implicit: such sequences will be referred to as *signals.* At a given instant, signals may have the status *absent* (denoted by $\perp$) and *present.* Jointly observed signals taking the status *present* simultaneously for any environment will be said to *possess the same clock,* and they will be said to possess different clocks otherwise. Hence clocks may be considered as equivalence classes of signals that are always present simultaneously. Operators of SIGNAL are intended to relate clocks as well as values of the various signals involved in a given dynamical system. Such systems have been referred to as *Multiple-Clocked Recurrent Systems (MCRS)* in [1]. To introduce the SIGNAL operators, we first discuss single-clocked systems, and then consider multiple-clocked ones.

2.1 Getting started in SIGNAL programming: simple examples

2.1.1 Monochronous signals: digital filtering

A classical second order digital filter is a representative for the class of dynamical systems having a single time index:

$$y_n = a_1 y_{n-1} + a_2 y_{n-2} + b_0 u_n + b_1 u_{n-1} + b_2 u_{n-2} \tag{1}$$

It allows us to introduce the operators of SIGNAL which handle what we will call *monochronous* (or synchronous) *signals,* i.e., signals with a common time index.

Such a filter is built from two types of equations:

1. $y_n = u_n + v_n$

2. $z_n = y_{n-1}$

Corresponding to these two types of equations, we have two types of *monochronous operators* in the SIGNAL language: the "static" ones and the "dynamic" one. Provided that the equations refer to the same index n, it is possible to make it implicit. Then the operators are defined on sequences of values (the *signals*).

Static monochronous operators are the extensions to sequences of the classical arithmetic or logical operators. Typical examples are `+`, `-`, `*`, `/`, `**`, `or`, `and`, `not`, `=`, `<`, etc. For instance, the SIGNAL equation

```
Y := U + V
```

is nothing but the coding of

$$\forall n \geq 0 \qquad y_n = u_n + v_n$$

with *implicit handling of the time index n.*

Dynamic monochronous operator: the delay

The SIGNAL delay operator defines the output signal whose n^{th} element is the $(n-k)^{th}$ element of the input one (k is a positive integer), at any instant but the first one at which it takes an initialization value. For example, the SIGNAL equation

```
Z := Y $1
```

is the coding of

$$\forall n > 0 \quad z_n = y_{n-1}$$

(the initial value y_0 is given in the declaration of `Z`).

An example of the behavior of the delay operator (with zero as initial condition) is shown in the following diagram:

`Y`:	2	5	1	0	4	1	3	7	9	...
`Z`:	0	2	5	1	0	4	1	3	7	...

To summarize, the `$k` operator corresponds to the z^{-k} shift operator used in signal processing or in control.

Composition of processes

SIGNAL equations such as those presented above define *elementary processes*; the *composition*

```
P1 | P2
```

of two processes P1 and P2 defines a new process, where common names refer to common signals (P1 and P2 communicate through their common signals). This is just the notion of conjunction of equations in mathematics. This operator is thus associative and commutative.

Defining $zy_n = y_{n-1}$, $zzy_n = zy_{n-1} = y_{n-2}$, ... makes the translation into SIGNAL of the filter (1) straightforward:

```
(| ZY := Y $1
 | ZZY := ZY $1
 | ZU := U $1
 | ZZU := ZU $1
 | Y := A1 * ZY + A2 * ZZY + B0 * U + B1 * ZU + B2 * ZZU |)
```

An alternative program uses the vector signals VY_n, VU_n, and constant vectors A and B:

$$VY_n = \begin{bmatrix} y_{n-2} \\ y_{n-1} \end{bmatrix} \quad , \quad VU_n = \begin{bmatrix} u_{n-2} \\ u_{n-1} \\ u_n \end{bmatrix} \quad , \quad A = \begin{bmatrix} A1 \\ A2 \end{bmatrix} \quad , \quad B = \begin{bmatrix} B0 \\ B1 \\ B2 \end{bmatrix}$$

Those vector signals are handled in SIGNAL with the following **window** operator:

```
VU := U window 3
```

defines a sliding window of length 3 on U.
The alternative program is then the following:

```
(| VY := Y $1 window 2
 | VU := U window 3
 | Y := PROD {A, VY} + PROD {B, VU} |)
```

with initial values given in the declarations of the vectors VY and VU. (PROD {V1, V2} is an externally defined function which computes the inner product of the vectors V1 and V2).

2.1.2 More advanced features

The *model* concept (or process declaration) encapsulates a set of equations; it allows the user to isolate local definitions and provides parameterized descriptions. A process model can be expanded (an instance of a model is a process).

Modular programming: block-diagrams

A graphical interface [5] has been designed to allow a user friendly definition of SIGNAL programs. A composition of processes has a hierarchical block-diagram representation (*parallelism* is thus a built-in concept in SIGNAL); the processes are represented by boxes; interconnections between input-output ports (or input-output signals) of the processes are represented by lines. The processes may be defined using equations or composition of equations (see figure 1), references to previously declared processes (see figure 4), or embedded graphical composition of processes.

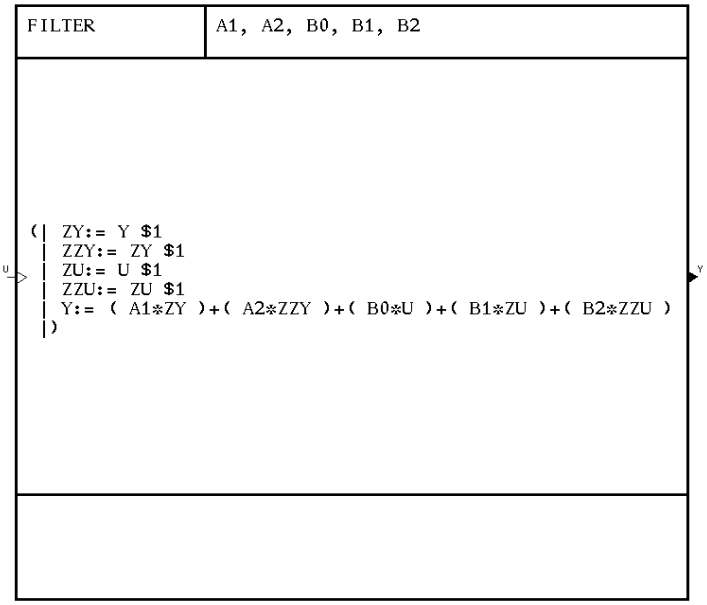


Figure 1: *A declaration of the process model* `FILTER`

The figure 1 depicts the graphical specification of the process model `FILTER` corresponding to equation (1). It is built using the SIGNAL graphical interface[1]. Note that `Y` is the only output signal visible from the outside of the process (the other ones are "local" signals).

Array of processes

The structure of "array of processes" is useful when specifying systolic algorithms or when describing regular architectures. As a simple example, the componentwise extension to vectors of a given operator may be defined by an **array** expression. For instance,

[1] In this paper, all block-diagram figures, except for figure 2, are copies of actual screens from the SIGNAL graphical interface.

```
array I to N of V := V1[I] * V2[I] end
```

is the extension of the product, as represented in the figure 2.

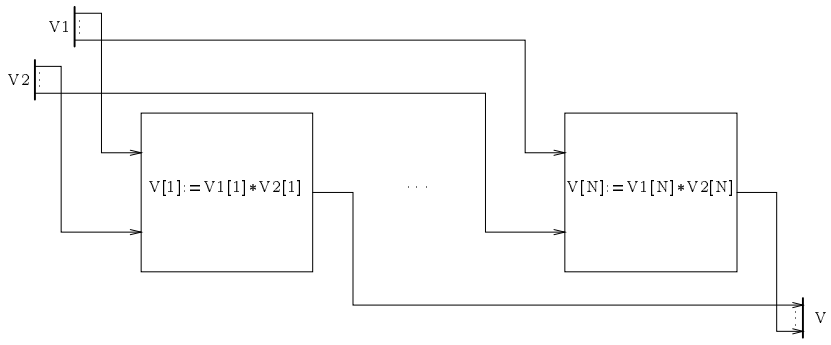


Figure 2: *An array of processes*

2.1.3 Summary

At this point, we are able to describe arbitrary dynamical systems possessing a single time index. Their coding is straightforward in SIGNAL. The modularity offered by the language is equivalent to that of signal flowgraphs or block-diagrams. Moreover, we can also describe regular arrays of processes.

Although these constructs are sufficient for classical digital signal processing or control, additional primitives are needed for developing complex real time applications. These will be introduced next.

2.2 Handling multiple-clocked systems

2.2.1 A small example: clicking on a mouse

We consider the mouse handler described in [1]. Let us recall its specifications. This process has two inputs:

- `CLICK`: a push-button,
- `TICK`: a clock signal.

The mouse handler has to repeatedly decide if, during some interval following an initial `CLICK`, some other `CLICK`s have been received; intervals are composed of a constant number $\Delta > 0$ of `TICK`s and are disjoint. At the end of each such interval, the mouse emits a signal `DOUBLE` when another `CLICK` has been received since the initial one, a signal

`SINGLE` otherwise. In [1], it has been discussed how this example may be specified using Multiple-Clocked Recurrent Systems (MCRS), see section 4.3 of this paper and equations (6-9) therein. From this discussion follows that two additional fundamental primitives are needed to specify such MCRS, namely:

- extracting a new time index from an existing one (equations (7,8,9) are instances of this),
- interleaving signals to produce the union of time indices (equation (6)).

The reader may also convince himself that these are convenient primitives; it has been argued in [2] that these are in fact the convenient primitives to provide a synchronous language with maximum expressive power for synchronization and control. These primitives are indeed primitive operators of SIGNAL. These are presented next.

2.2.2 Polychronous operators

The extraction: the SIGNAL process

```
Y := X when B
```

where `X` and `Y` are signals and `B` is a boolean signal, delivers `Y` = `X` whenever `X` and `B` are present and `B` is *true*, and delivers nothing otherwise. The behavior of the `when` operator is illustrated in the following diagram:

`X` :	1	2	$\perp$	3	4	$\perp$	5	6	9	...
`B` :	*t*	*f*	*t*	*f*	*t*	*f*	$\perp$	*f*	*t*	...
`Y` :	1	$\perp$	$\perp$	$\perp$	4	$\perp$	$\perp$	$\perp$	9	...

($\perp$ stands for "no value"). The **when** operator may be proved associative and idempotent in the set of events. When `X` is a constant, the clock of `X when B` is the clock of `B when B`.

The deterministic merge: the SIGNAL process

```
Y := U default V
```

defines `Y` by merging `U` and `V`, with priority to `U` when both signals are present simultaneously. It yields `Y` = `U` whenever `U` is available, and `Y` = `V` whenever `V` is available but `U` is not; otherwise, `Y` is not delivered. The behavior of the `default` operator is illustrated in the following diagram:

`U` :	1	2	$\perp$	3	4	$\perp$	5	$\perp$	9	...
`V` :	$\perp$	$\perp$	3	4	10	8	9	2	$\perp$	...
`Y` :	1	2	3	3	4	8	5	2	9	...

The `default` operator may be proved associative (which avoids the use of parentheses). Moreover, `when` is right distributive on `default`. When `V` is a constant, the clock of `Y` is any clock greater than the clock of `U`.

2.2.3 Some extensions

When specifying time constraints, it may be useful to refer to the clock of some signal. The following derived operators are of particular interest in that case.

- The variation

  ```
  T := when B
  ```

  of the `when` operator defines the **event** type signal `T` which is present whenever the boolean signal `B` is present and has the value *true* and delivers nothing otherwise; it is equivalent to `T := B when B`. An **event** type signal (or "pure" signal) is an always *true* boolean signal. Hence `not T` denotes the boolean signal with clock `T` which always carry the value *false*.

- Given any signal `X`,

  ```
  T := event X
  ```

  defines the **event** type signal `T` whose occurrences are simultaneous with those of `X`: it represents the clock of `X`.

- Finally constraints may be defined on the clocks of signals. In this paper, the following notations are used:

  `X ^= Y` `X` and `Y` have the same clock [2];

  `X ^< Y` `X` is no more frequent than `Y`, which is equivalent to `X ^= (X when event Y)`.

The following derived operator specifies a synchronized memory: the SIGNAL process

```
Y := X cell B
```

where `B` is a boolean signal, delivers at the output `Y` either the present value of `X` (when the latter is present), or the last received value of `X` when `B` is present and *true*. It is equivalent to:

```
(| Y := X default (Y $1)
 | Y ^= (event X) default (when B) |)
```

[2] it is written `synchro {X, Y}` in the syntax of the current version

2.2.4 Programming the mouse

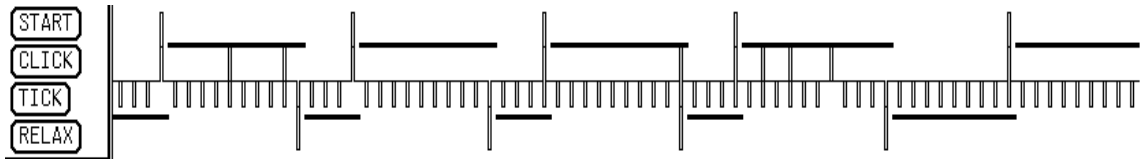


Figure 3: *A chronogram of the mouse*

A "chronogram" of the mouse is described in the figure 3[3]. This shows the sequence of intervals where CLICKs are monitored (in the figure, the number of TICKs in an interval is $\Delta = 10$). As it appears in the figure, we introduce naturally the two following pure signals:

- START, which indicates the beginning of a new interval,
- RELAX, which indicates the end of the current interval.

Then, consider a first module which aims at producing the outputs of the MOUSE, namely SINGLE and DOUBLE. This module gets as its inputs CLICK, START and RELAX. The corresponding specification is:

```
(| DOUBLE_CLICK := ((not START) default (CLICK in ]START, RELAX])) cell RELAX
 | SINGLE := RELAX when (not DOUBLE_CLICK)
 | DOUBLE := RELAX when DOUBLE_CLICK |)
```

The meaning of these equations is the following. DOUBLE_CLICK is a boolean signal which states at the end of the elapsed time whether a single (status $false$) or several (status $true$) CLICKs have been received. For this purpose, each START sets DOUBLE_CLICK to $false$ (not START is taken with priority). Since STARTs are also CLICKs, at least one CLICK has been received in the considered interval. Then if a second CLICK is received within the allowed delay, DOUBLE_CLICK is set to $true$. Testing for this is performed by the expression "CLICK in]START, RELAX]" defined as follows:

```
X in ]S,T]                (i)
```

delivers those X's which belong to the left-open and right-closed interval]S,T], where S and T are both pure signals. Note the cell RELAX expression which delivers at every RELAX the current status of DOUBLE_CLICK.

What remains now is to indicate how to produce the events START and RELAX. For this purpose, two operators are introduced:

```
X not in ]S,T]            (ii)
#X in ]S,T]               (iii)
```

[3]this figure depicts a simulation environment for the mouse written in SIGNAL under SunView

Expression (ii) delivers those X's which do not belong to]S,T]. Expression (iii) counts the occurrences of X within the mentioned interval and is reset to zero every S; this signal is delivered exactly when equation (i) delivers its output. Using these operators, the second module of the MOUSE program is presented next:

```
(| START := CLICK not in ]START, RELAX]
 | (| N := (#TICK in ]START, RELAX]) cell event N
    | ZN := N $1          % initial value 0 %
    | N ^= CLICK default TICK
    | RELAX := TICK when (ZN = (DELTA-1)) |)
 |)
```

The first equation selects those CLICKs that are also STARTs, and selects also the first CLICK. The other equations count the TICKs and deliver the result as frequently as needed (thanks to cell event N). A graphical editing of the resulting MOUSE program is shown in the figure 4 using the SIGNAL graphical interface. In this figure, the two modules we introduced are labelled SIMPLE_MOUSE and GO respectively. Note that CLICK and TICK are independent inputs of this program.

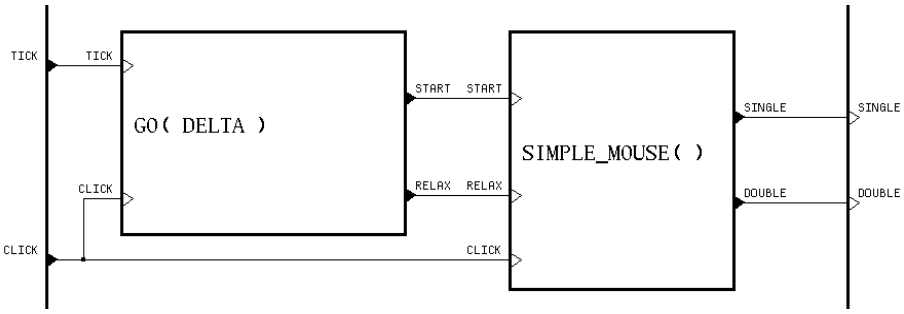


Figure 4: *The process model* MOUSE

Comments: the text of the two above modules should be taken as a specification since the operators we introduced are not available in the current version of the language. They will be available however in a forthcoming version of it, with all variations on the shape of the considered interval ([S,T[, [S,T], etc.). Thus we shall present without further discussion this program written in the current version of SIGNAL where these macros are built as SIGNAL processes. Then we shall provide the expansion in SIGNAL of the operator (i).

The actual program is the following:

```
process MOUSE = (integer DELTA)
   { ? event TICK, CLICK
     ! event SINGLE, DOUBLE }
   (| (| START := NOT_IN_INTERVAL {CLICK, START, RELAX}
      | (| N := COUNT_IN_INTERVAL {TICK, START, RELAX} cell event N
         | ZN := N $1
         | N ^= CLICK default TICK
         | RELAX := TICK when (ZN = (DELTA-1)) |)
    | (| DOUBLE_CLICK := ((not START) default IN_INTERVAL {CLICK, START, RELAX})
                         cell RELAX
      | SINGLE := RELAX when (not DOUBLE_CLICK)
      | DOUBLE := RELAX when DOUBLE_CLICK |)
    |)
   where event START, RELAX; integer N, ZN init 0; logical DOUBLE_CLICK
end
```

The first three lines specifie the name of the process model and its interface (`DELTA` is a parameter; `?` stands for "input" and "!" for "output"). `IN_INTERVAL`, `NOT_IN_INTERVAL` and `COUNT_IN_INTERVAL` are instances of subprocesses corresponding respectively to the operators `(i)`, `(ii)` and `(iii)` presented above.

As an example, the process `IN_INTERVAL`, corresponding to the expression `X in ]S,T]`, may be defined as follows:

```
process IN_INTERVAL = { ? X; event S, T
                        ! Y }
   (| BELONGS_TO_INTERVAL ^= (S default T default (event X))
    | (| WILL_BELONG := (not T) default S default BELONGS_TO_INTERVAL
       | BELONGS_TO_INTERVAL := WILL_BELONG $1 |)
    | Y := X when BELONGS_TO_INTERVAL
    |)
   where logical WILL_BELONG, BELONGS_TO_INTERVAL init false
end
```

Processes `NOT_IN_INTERVAL` and `COUNT_IN_INTERVAL` corresponding to operators `(ii)` and `(iii)` are defined similarly.

Using SIGNAL for specifying a *Multiple Clocked Recurrent System* [1] releases the programmer from the burden of handling explicitly multiple time indices. *Every signal in the language has an implicit time index and the* SIGNAL *operators define relations between the time indices.*

2.3 Summary: SIGNAL-kernel

To summarize, the kernel-language SIGNAL possesses only five basic constructions which are recalled here:

`Y := f(X1,...,Xn)`	extending instantaneous functions to signals with common clock
`Y := X $N`	delay (shift register)
`Y := X when B`	condition based downsampling
`Y := U default V`	merge with priority
`P | Q`	composition of processes

All other operators are built as macros on this kernel-language and model declarations. Moreover the language allows modular programming and external functions calls. It can be used to describe internally or externally generated interruption or exception handling, data-dependent down- and upsampling [3], mixed passive/active communications with the external world [4]. Thus the SIGNAL language has all the features needed for real time applications programming. It has been proved in [2] that SIGNAL possesses maximum expressive power for synchronization mechanisms, in particular *data dependent upsampling* can be expressed in SIGNAL which proved very useful in most of the applications we developed.

The following feature of SIGNAL programming style should be emphazised. Since the compiler synthesizes the global timing from the programmer's specifications, the following programming style is recommanded: *specify local timing constraints involving very few different clocks, and let the compiler do the rest.* This is different from LUSTRE's programming style, where the programmer must have a global view of the timing to write the program.

2.4 Specifying logical temporal properties

Various techniques are used to verify programs: *temporal logic* [6, 13] in CSML and the STATECHARTS, automata reductions and verification [14, 15] in ESTEREL and LUSTRE for instance. The LUSTRE language also uses assertions to express constraints on boolean signals, and offers tools to compute boolean dynamical expressions written in LUSTRE itself [8]. Thanks to the powerful model of SIGNAL, the SIGNAL language itself can be used as a partial proof system.

As an example, consider a memory `M`, which can be written (signal `WRITE`) and read (signal `READ`):

```
(| M := WRITE default (M $1)
 | READ := M when (event READ) |)
```

Each value written in `M` (first line) is read when needed (second line).

Now suppose that writing in the memory is allowed only when the previous value of the memory has been read. Let us encode the status (being written or being read) of the memory as follows:

```
FULL := (event WRITE) default (not (event READ))
```

Then the above constraint is expressed by the following equation:

```
WRITE ^= when (not (FULL $1))
```

Conversely, if we want any written value to be read at most once, we have to write:

```
READ ^= when (FULL $1)
```

Finally, putting these three additional equations together *specifies* a single token buffer, it turns out that this is also its *programming*.

This example illustrates an important feature of the SIGNAL language. To insure that a property is verified on a SIGNAL program, encode this property as SIGNAL equations. This equations may be used in different ways. First it could be checked whether the corresponding constraints are already implied by the program. Second the equations may be simply added to the program to make sure that the desired property be satisfied. We will see in the next section how SIGNAL's "clock calculus" can be used for this purpose.

3 The SIGNAL compiler as a formal calculus system

3.1 The formal model

The reasoning mechanisms of SIGNAL handle (i) the presence/absence, (ii) boolean values since they are important in modifying clocks, and (iii) the *dependency graphs* to encode data dependencies in non-boolean functions. Dependency graphs are needed to detect short circuits such as in `X := X+1`, and to generate the execution schemes. Three labels are used to encode *absent, true, false* as well as the status *present* we consider as a non-determinate "$true$ or $false$" value. The finite field $\mathcal{F}_3$ of integers modulo 3 is used for this purpose[4]:

$$true \to +1,\ false \to -1,\ absent \to 0,\ present \to \pm 1$$

For instance, using this mapping, `(a or b) = event a` and `y := u+v` are respectively encoded as follows:

$$a^2 = b^2 \quad , \quad ab(a-1)(b-1) = 0 \tag{2}$$

$$y^2 = u^2 = v^2 \quad , \quad \mathtt{u} \xrightarrow{y^2} \mathtt{y} \quad , \quad \mathtt{v} \xrightarrow{y^2} \mathtt{y} \tag{3}$$

In these equations, the variables $a, b, \ldots$ refer to infinite sequences of data in $\mathcal{F}_3$ with time index implicit. The first equation of (2) expresses that the two signals `a` and `b` must have

[4] elements of $\mathcal{F}_3$ are written $\{-1, 0, 1\}$

the same clock, while the second one encodes the particular boolean relation. The first equation of (3) again expresses that all signals must have the same clock, while the labelled graph expresses that the mentioned dependencies hold when $y^2 = 1$, i.e., when all signals are present. This is referred to as the *conditional dependency graph*, since signals may be related via different dependencies at different clocks. Let us describe how the other primitive operators of SIGNAL are encoded in this way.

Process `y := x $1`.
As easily checked, boolean shift registers are encoded as follows:

$$\begin{aligned} \xi_{n+1} &= (1-x^2)\xi_n + x \quad , \quad \xi_0 = \texttt{yo} \\ y &= x^2\xi_n \end{aligned}$$

In this equation, ξ_n is the current state, and ξ_{n+1} is its next value according to any (hidden) clock which is more frequent than the clock of x ($\xi_0 = \texttt{yo}$ is the initial value). This is a nonlinear dynamical system over $\mathcal{F}_3$. The non-boolean shift register is just encoded via the equality of clocks: $y^2 = x^2$.

Process `y := x when b`.
In the boolean case, we get the coding

$$y = x(-b - b^2)$$

while in the non-boolean case, we must encode the constraints on clocks and dependencies:

$$y^2 = x^2(-b - b^2) \quad , \quad \texttt{x} \xrightarrow{y^2} \texttt{y}$$

Process `y := u default v`.
In the boolean case we get

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

while in the non-boolean case we get:

$$y^2 = u^2 + v^2(1 - u^2) \quad , \quad \texttt{u} \xrightarrow{u^2} \texttt{y} \quad , \quad \texttt{v} \xrightarrow{(1-u^2)v^2} \texttt{y}$$

Process `P | Q`.
Here `P`, `Q` denote SIGNAL processes. The graph of the process `P|Q` is the union of graphs of `P` and `Q`; in the same way, the equations associated with the process `P|Q` are the equations of `P` and those of `Q`.

Moreover, in addition to dependencies between signals, dependencies relating signals and

clocks must be considered. In particular, any signal `y` depends on its clock y^2, as expressed by the dependency:

$$y^2 \xrightarrow{y^2} \texttt{y}$$

Finally we end up with the general form to encode any SIGNAL program:

$$\left\{ \begin{array}{rcl} \Xi_{n+1} & = & A(\Xi_n, Y_n) \\ 0 & = & B(\Xi_n, Y_n) \\ 0 & = & C(\Xi_0, Y_0) \end{array} \right.$$

$$\texttt{Y(i)} \xrightarrow{H(i,j)} \texttt{Y(j)} \quad , \quad Y(i)^2 \xrightarrow{Y(i)^2} \texttt{Y(i)} \tag{4}$$

In this system, Ξ, Y are vectors with components in $\mathcal{F}_3$, A, B, C denote polynomial vectors on the components $\Xi(i), Y(j)$ of Ξ, Y. The components of Ξ are the states of the boolean registers, and the components $Y(j)$ of Y are the encoding in $\mathcal{F}_3$ of all signals `Y(j)` involved in the program. The time index n may be any time index which is more frequent than the clocks of all components of `Y`. The two last equations specifie the conditional dependencies, where $H(i,j) = 1$ specifies the clock where the referred dependency holds. The equations (4) show why the work of the SIGNAL compiler relates to formal calculus on dynamical systems involving the finite field $\mathcal{F}_3$ and graphs.

It is shown in [3, 4, 9] how this coding can be used, with the help of polynomial ideal theory, to answer fundamental questions about the properties of a given program:

1. *Does the program exhibit contradictions?* Consider for instance the following program:

```
(| x := a when (a > 0)
 | y := a when not (a > 0)
 | z := x + y |)
```

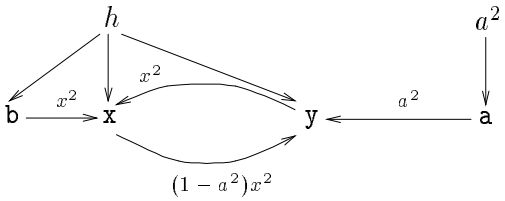
   Writing α for short instead of `(a > 0)`, its clock calculus yields $-\alpha - \alpha^2 = \alpha - \alpha^2$ whence $\alpha = 0$: this means that `a` must be always absent, the program refuses its inputs and does nothing.

2. *Are there short circuits?* Consider the following program:

```
(| x := sin {y} + b
 | y := a default x |)
```

   The clock calculus and conditional dependency graph are

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



Due to the short circuit including `x` and `y`, this program is deadlocked unless the clock of this short circuit is always absent, i.e., $(1-a^2)x^2 = 0$, or equivalently, $(1-a^2)b^2 = 0$. Hence, $y^2 = a^2$, and this program implements:

```
(| y := a
 | x := sin {a} + b |)
```

3. *Is the program setting constraints on its inputs?* Consider the program:

```
(| x := a when (a > 0)
 | z := a + x |)
```

Writing α instead of `(a > 0)`, the clock calculus is

$$z^2 = a^2 = x^2 \quad , \quad x^2 = a^2(-\alpha - \alpha^2) \quad , \quad \alpha^2 = a^2$$

which forces

$$\alpha^2 = 0 \quad \text{or} \quad 1 + \alpha + \alpha^2 = 0 \ \text{ i.e. } \ \alpha = 1$$

Hence when `a` is present, we *must* have `a > 0` otherwise the program is deadlocked by a contradiction. However SIGNAL cannot reason on non-boolean data types. Hence, considering that α is the output of a non-boolean function (testing `a > 0`), the constraint $\alpha^2(1-\alpha) = 0$ is replaced by the stronger one $\alpha^2 = 0$, which does not involve the *value* ($true$ or $false$) of α any more: `a` is then refused so that this program refuses to do anything.

4. *Is the program deterministic, i.e., is it a function?* Consider the following program (which specifies a counter with external reset):

```
process P = { ? s ! t }
   (| nt := (0 when s) default (t+1)
    | t := nt $1 |)
end
```

Its clock calculus yields

$$nt^2 = t^2 = s^2 + (1-s^2)t^2$$

which is equivalent to $t^2 \geq s^2$: if `s` is the specified input, the clock of the output `t` is not a function of any external signal. Hence this program is not a function. Inserting the following synchronization equation, `t ^= (s default u)`, where `u` is another input) completely specifies the timing and we get a function.

5. *Does the program verify some property?*—the specification of the buffer presented in section 2.4 is a good exercise.

3.2 The work of the compiler

We have briefly described the mathematical model supporting the work of the compiler. The way the compiler uses this model is the following. The compiler uses a very efficient algorithm to construct a *hierarchy* of clocks with respect to the following rules:

- If `C` is a *free* boolean signal (i.e., it results from the evaluation of a function with non-boolean arguments, or it is an input signal of the program, or it is the status of a boolean memory), then the clock defined by the *true* values of `C` (i.e., `when C`) and the clock defined by the *false* values of `C` (i.e., `when not C`) are put under the clock of `C`; both are called *downsamplings*.
- If a clock `K` lies under a clock `H` then every clock which lies under `K` also lies under `H`.
- Let `H` be a clock defined as a function of *downsamplings* $\mathtt{H}_1,\ldots,\mathtt{H}_n$, if all these *downsamplings* lie under a clock `K`, then `H` also lies under `K`.

The resulting hierarchy is a collection of interconnected trees, say a *forest*. The partial order defined by this forest represents *dependencies* between clocks: the actual value of a clock `H` may be needed to compute the actual value of a given clock `K` only if `H` lies above `K` according to this partial order. No hierarchy is defined on the roots of the trees, but constraints can exist. When this forest reduces to a single tree, then a single master clock does exist, from which other clocks derive. In this latter case, the program can be executed in master mode, i.e., by requiring the data from the environment. If several trees remain, additional synchronization has to be provided by the external world (e.g. small real time kernels, see [1]) or by another SIGNAL program.

The conditional dependency graph is attached to the forest in the following way. The signals available at a given clock are attached to this clock, and so are the expressions defining these signals. The so obtained *"conditional hierarchical graph"* is the basis for sequential as well as parallel code generation.

Moreover, the proper syntax of SIGNAL can be used to represent this graph. For that purpose, the compiler rewrites the clock expressions as SIGNAL boolean expressions: the operator `default` represents the upper bound of clocks (sum) and the operator `when` represents the lower bound (product); then, any clock expression may be recursively reduced to a sum of monomials, where each monomial is a product of downsamplings (otherwise, the clock is a root). The definitions of the signals are also rewritten to make explicit the clocks of the calculations that define these signals.

The rewritten process is equivalent to the initial one, but the clock and dependency calculus is now solved, and all the clocks handled in the program are made precisely explicit. The so obtained process will be referred to as the *solved* form of the considered program.

An example taken from the MOUSE is developed in the appendix. Its solved form, which exhibits a forest of several clock trees, is detailed. Then, a simulated real-time monitor is provided which delivers the inputs CLICK and TICK to this program. This simulator is itself written in SIGNAL. The pair {program, monitor} is then processed by the compiler and produces a single tree for its solved form. This solved form is shown and the sequential C code generated from this program is given.

4 Toward parallel implementation

A distributed implementation of a SIGNAL program P consists of a definition of P as

$$\mathtt{P} \;=\; (|\; \mathtt{P}_1 \;|\; \ldots \;|\; \mathtt{P}_n \;|)$$

into *modules* $\mathtt{P}_1$, ..., $\mathtt{P}_n$ which will be one to one mapped onto a set of n processors. Thanks to the equational approach, the modules $\mathtt{P}_i$ can be built either downwards by breaking, or upwards by clustering subprocesses. Hence we have developed a systematic method to serialize such modules, while avoiding possible deadlocks. This method, which generalizes the use of semi-granules such as presented in [10], is outlined next. It turns out that the same method can be used to improve the efficiency of the implementation, by reducing the overhead due to process scheduling.

4.1 Some issues on distribution

The following notations will be used for the figures throughout this section: solid arrows denote data dependencies enforced by the considered programs, dashed arrows indicate additional ordering that results from a given implementation. For instance, in figure 5-a, the program specifies that a must be received first before producing x (and similarly for b and y), and the dashed arrows express that in the considered implementation, it is first waited for *both* a and b, and then x and y are produced. Adding dashed lines within a dependency graph will be referred to in the sequel as performing *order enhancement.*

Using these notations, consider the following program, where f and g are some arbitrary functions:

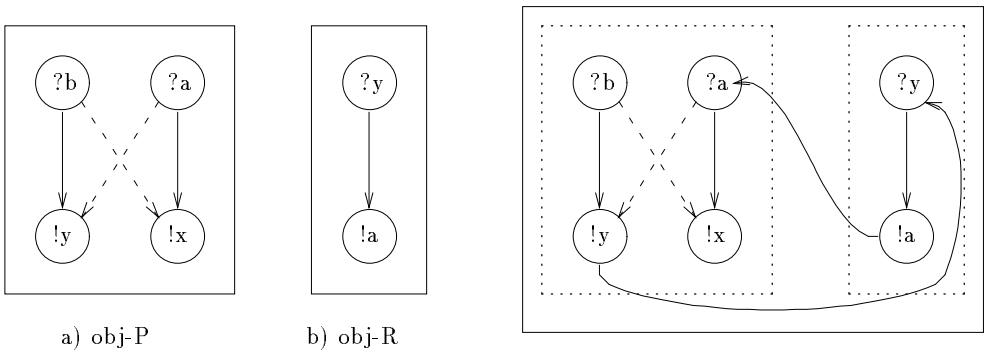
```
P = (| y := g(b) | x := f(a) |)
```

The sequence of getting values followed by putting results, repeated forever, is a correct execution scheme of P if we assume that any input signal is available whenever needed; each step is described in figure 5-a.

Unfortunately, the context of P may for instance be the following SIGNAL program:

```
R = (| a := h(y) |)
```

where h is again some function. Its only correct execution scheme is the sequence of getting y followed by putting a, repeated forever as described in figure 5-b.



a) obj-P b) obj-R c) Deadlock

Figure 5: *Context dependent implementation*

The SIGNAL source program `P|R` is certainly a correct one. However, the concurrent[5] execution of their sequential implementation `obj_P` and `obj_R`, is obviously deadlocked (figure 5-c): `obj_P` is waiting for `a`; to produce `a`, `obj_R` needs `y` which cannot be delivered by `obj_P`. This is depicted by the cycle in the figure 5-c. Now if we consider the following program (see figure 6-a):

```
Q = (| y := g(a,b) | x := f(a,b) |)
```

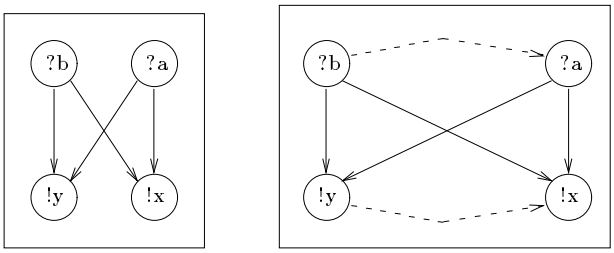
then for any program `R'` such that `y` or `x` is needed to calculate `a` or `b`, the program `Q | R'` is incorrect. Thus any implementation of this program `Q` in which communications are serialized in agreement with the local partial order specified by the graph of figure 6-a is a correct one. For instance, sequence of {getting `b` ; getting `a` ; putting `y` ; putting `x`} repeated forever does not cause additional deadlocks whatever the environment is. This implementation `obj_Q` is depicted by the added dashed lines in figure 6-b.

This is what we call *order enhancement* of the graph. Thus the key to code distribution is the *dependency graph,* and possible deadlocks with the environment that might result from an unclever order enhancement must be prevented. Appropriate tools for the general case of multiple clocks are briefly presented in the next section.

4.2 Conditional dependency graph, interface conditional graph, and code distribution

Motivated by the discussion of this simple example, we present now the following method for code distribution. We assume that the distribution of the graph of the program has been performed according to suitable criteria we don't consider here. Then we concentrate

[5] in the sense of multitasking systems



a) Q b) obj-Q

Figure 6: *Second example*

on one particular module. For this module, the method consists of the three following stages:

1. calculate transitive dependencies of external signals: this yields the *interface conditional graph;*
2. given this interface conditional graph, calculate all *legal* order enhancements (that are guaranteed compatible with any arbitrary correct environment);
3. from these legal order enhancements, calculate a proper execution scheme of the considered module.

The so-obtained object code can be stored as a reusable executable module. Steps 1, 2, 3 are also the way to separate compilation of modules.

4.2.1 Getting the interface conditional graph

It is easily derived using the two following rules:

$$\text{rule of series} \qquad \mathtt{X} \xrightarrow{h} \mathtt{Y} \xrightarrow{k} \mathtt{Z} \;\Rightarrow\; \mathtt{X} \xrightarrow{hk} \mathtt{Z} \tag{5}$$

(`X` precedes `Z` whenever `X` precedes `Y`, at the instants where $h = 1$, *and* `Y` precedes `Z`, at the instants where $k = 1$).

$$\text{rule of parallel} \qquad \left.\begin{array}{l} \mathtt{X} \xrightarrow{h} \mathtt{Y} \\ \mathtt{X} \xrightarrow{k} \mathtt{Y} \end{array}\right\} \;\Rightarrow\; \mathtt{X} \xrightarrow{h \vee k} \mathtt{Y} \tag{6}$$

where $h \vee k = h + (1-h)k$ denotes the supremum of the two clocks h and k (h and k are polynomial functions in $\mathcal{F}_3$ taking $0, 1$ as only values): `X` precedes `Y` whenever `X` precedes `Y` at the instants where $h = 1$, *or* `X` precedes `Y` at the instants where $k = 1$.

Successive applications of these rules yield the kind of graph depicted as solid branches in the figure 7 (in which local nodes do not appear).

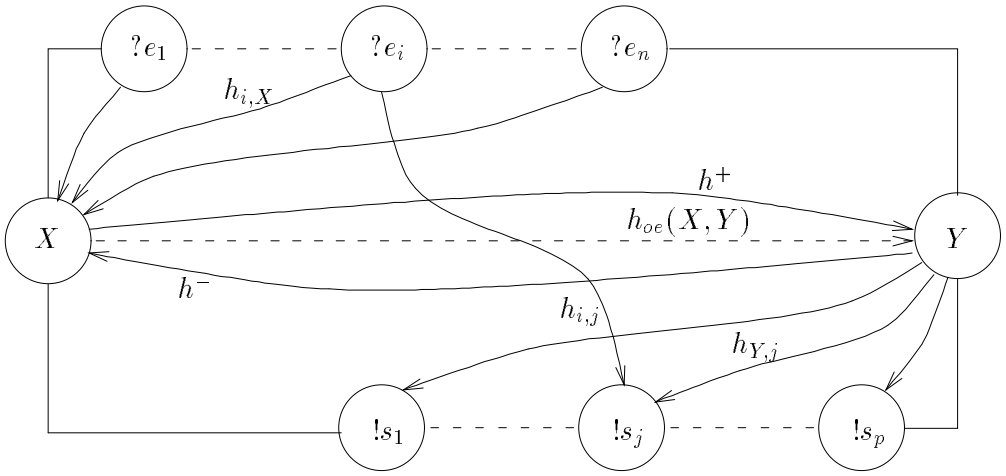


Figure 7: *Order enhancement*

4.2.2 The legal order enhancements

Referring to the figure 7, let us concentrate on two interface signals, say `X` and `Y`. Denote generically by $h_{oe}(\mathtt{X},\mathtt{Y})$ the clock of some legal order enhancement that puts `X` before `Y` in the execution scheme. The conditions which must be satisfied by $h_{oe}(\mathtt{X},\mathtt{Y})$ are the following:

1. No internal cycle should result from the additional clock $h_{oe}(\mathtt{X},\mathtt{Y})$ in the graph. This yields the condition:
$$h_{oe}(\mathtt{X},\mathtt{Y})\ h^- = 0 \tag{7}$$

2. No possibility of an additional cycle due to the environment results from $h_{oe}(\mathtt{X},\mathtt{Y})$; this yields the inequalities:
$$\forall i,j \quad h_{i,X}\ h_{oe}(\mathtt{X},\mathtt{Y})\ h_{Y,j} \ \leq\ h_{i,j} \tag{8}$$
(every input $\mathtt{e}_i$ which precedes `X` also precedes every output $\mathtt{s}_j$ following `Y`: this insures that, in any context, no dependency from an output $\mathtt{s}_j$ to an input $\mathtt{e}_i$ can be introduced, which could create a deadlock).

Elementary algebra shows that (7,8) can be summarized as the single inequality:

$$h_{oe}(\mathtt{X},\mathtt{Y}) \;\leq\; h^+ + X^2Y^2(1-h^- -h^+)\prod_{i,j}(1-h_{i,X}h_{Y,j}(1-h_{i,j})) \qquad (9)$$

We will say that a conditional dependency graph G_1 is lower than another one G_2 if and only if they have the same nodes, and each time $x \rightharpoonup y$ occurs in G_1 (when its label h_1 is equal to 1), then $x \rightharpoonup y$ occurs in G_2 ($h_2 = 1$); so $h_1 \leq h_2$. Applying order enhancement results in a graph where each $h_{oe}(\mathtt{X},\mathtt{Y})$ takes its maximal value (it is not the graph of a partial order but the upperbound of the maximal order enhancements).

4.2.3 Getting execution schemes

Consider again the program `P` above, and denote by h the clock of all solid branches in the figure 5-a. The original graph coincides with the interface conditional graph, and the formula (9) shows that no legal order enhancement does exist in this case, so that the only reusable form is the source code.

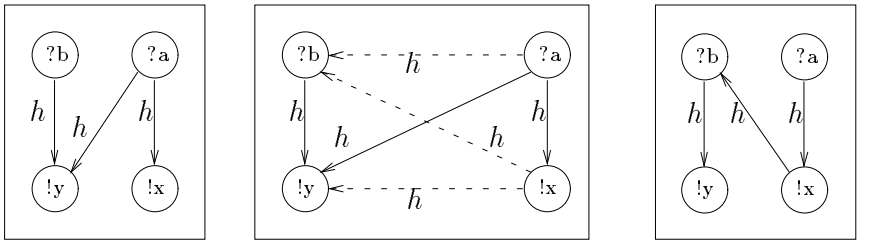


Figure 8: *Sequential order enhancement*

Now, consider some program `S` whose conditional dependency graph is shown in the figure 8-a ; the resulting order enhancement is depicted in the figure 8-b. `S` has the unique sequential execution scheme shown in the figure 8-c. It is obtained by picking the subgraph of the dashed or solid branches that is both a path and covers all nodes.

For some programs, the order enhancement may result in a cyclic graph as shown in the figure 9-b. Such cycles do not express that deadlocks have been created, but just indicate that external communications within the cycle can be performed in an arbitrary order, depending on the environment's offer or request at a particular instant. For instance, we may equally well first receive `a` and then `b` or the converse: this is depicted in the figure 9-c.

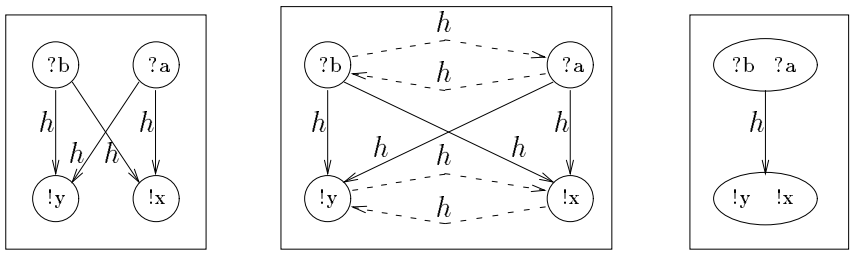


Figure 9: *Cyclic order enhancement*

4.2.4 The lazy evaluation of a module

Similar techniques may be used to calculate the clock h_Z of those instants where it is really needed to compute a signal `Z` at the execution: `Z` must be computed when it is needed to compute some output of the module or some state variable, and the corresponding clock is calculated using the "rule of series" (5) and "rule of parallel" (6) we have shown before.

4.2.5 Getting a methodology for distributed implementations

From the discussion above emerges the following method:

- Separate compilation may be performed following the method we outlined above: synthesizing the interface conditional graph, and then deducing the scheduling from the order enhancements yields a control process `C` associated with a given program `P`, this module can then be used as executable code in any environment.
- Alternatively, it is also possible to *specialize* this control process using some prior information on the environment (e.g. other SIGNAL modules or the properties of their interfaces) that are also stated in suitable control processes.

5 Programming environment

We present here a realistic experience with SIGNAL, which has been used to describe the acoustic-phonetic decoder of an automatic speech recognition system. Our purpose is not to detail the program (which would be much too long—the interested reader is referred to [12]), but rather, to give a flavour of how a large project could be developed with the SIGNAL environment.

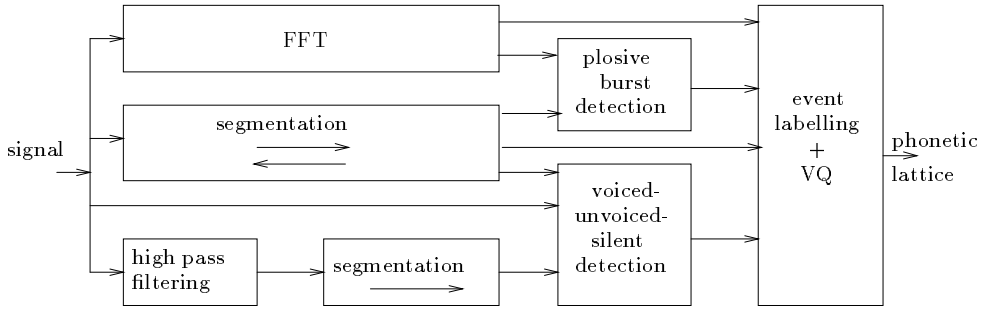


Figure 10: *Modular description of an acoustic-phonetic decoder*

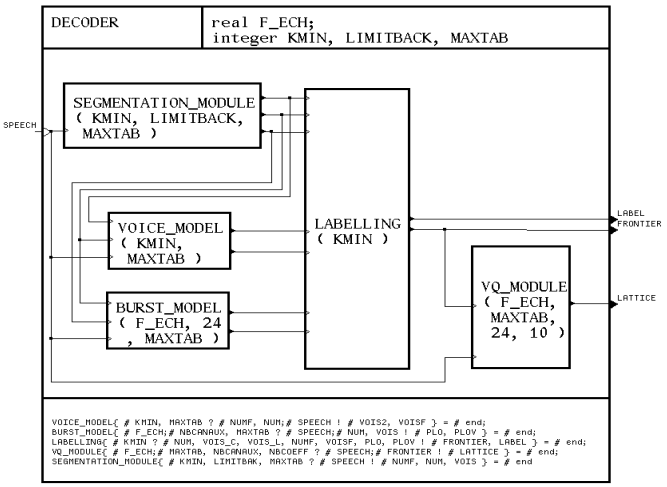


Figure 11: *A graphical view of the* `DECODER` *process model which is composed of an automatic segmentation (*`SEGMENTATION_MODULE`*), a voiced-unvoiced-silent decision (*`VOICE_MODEL`*), a detection of plosive bursts (*`BURST_MODEL`*), a coordination between boundaries labelling (*`LABELLING`*) and vector quantization (*`VQ_MODULE`*)*

5.1 A speech-to-phoneme recognition system: global description

The reader is referred to [1] for a more detailed description of this application. The figure 10 depicts a block-diagram of a part of the speech-to-phoneme recognition system as developed at IRISA. The `FFT` box involves a sliding-block processing of the speech signal. The `filtering` and `segmentation` boxes process the speech signal sample-by-sample. The $\rightarrow$ (resp. $\overrightarrow{\leftarrow}$) inside the segmentation boxes indicates that the signal is processed forward only (resp. forward-backward): the data-dependent upsampling mechanism is used in the corresponding SIGNAL programs. The `detection` and `event labelling` boxes involve event detection. Thus several sophisticated mechanisms that are provided by SIGNAL

were used in this application. We should emphasize that the IRISA speech group was reluctant to write any real time oriented FORTRAN programming of this application, only SIGNAL allowed us to develop such a real time programming. Finally, the SIGNAL graphical interface proved well suited for developing this application. The figure 11 shows a graphical view of the decoder as written in SIGNAL.

5.2 Building a control panel for experimentation

To take advantage of the SIGNAL approach, a tool-box for the on-line scanning of the results has been developed using SIGNAL. These developments were intended to allow an on-line interaction of the user during the execution, with both the program itself and the display of its results. This is achieved without modifying the source program, but just by connecting "probe" and "debug" modules we describe briefly:

- "probe" processes allow to monitor the program without disturbing its execution. Such a process is associated with a port of the program. The figure 12 shows a probe process associated with the speech signal. A probe process is a SIGNAL process with no output, which is declared as an external process to be analyzed by the display system (X-windows or SunView).

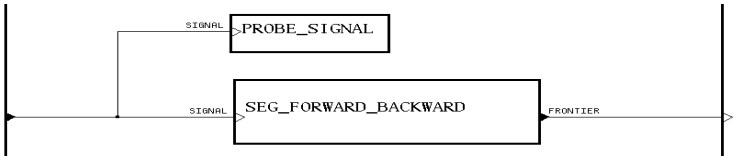


Figure 12: *A probe process is associated with the speech signal*

- "debug" processes allow to control the running of the program through a panel-driven down- or upsampling of some signals, or the on-line change of some parameters. Such a process is associated with a link between two ports (figure 13).

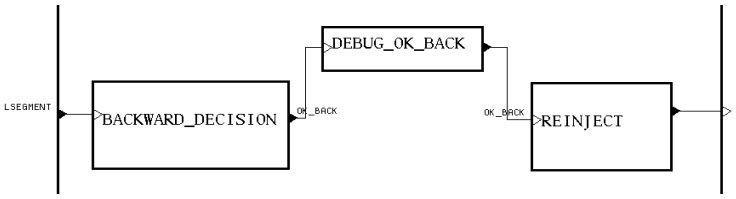


Figure 13: *A debug process is associated with the backward signal*

- An intermediate tool consists of a "pace maker", which makes only the program running slower by encapsulating it in a program accessing a physical clock. The logical time may be a subset of this clock managed by up and down buttons.

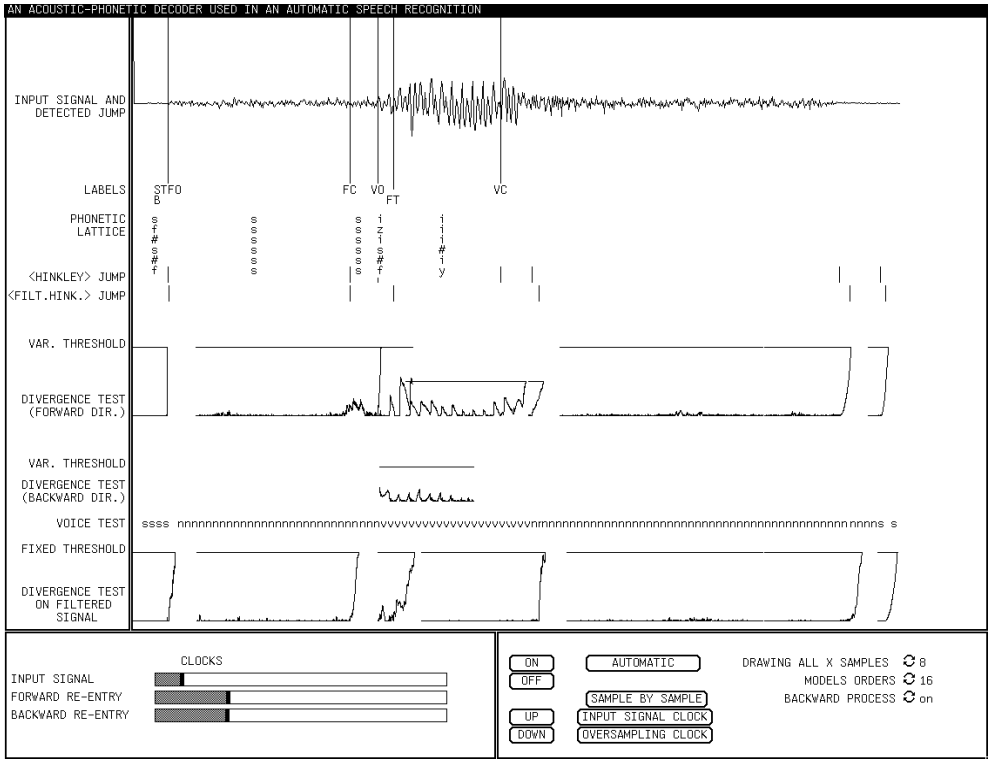


Figure 14: *Synchronous environment for an acoustic-phonetic decoder (pronounced digit: "6")*

The figure 14 shows an environment for the acoustic-phonetic decoder, developed under the SunView window management system.

6 Conclusion

We have presented the SIGNAL synchronous programming language for real time systems development. The following key features should be mentioned:

- SIGNAL is a block-diagram oriented language. As such, it is provided with a graphical interface for program editing and execution.
- Since block-diagrams naturally specify constraints or relations between the involved signals, SIGNAL is a language of *equational* style. This has several important consequences we list now:
  - The programmer has only to specify *local* synchronization constraints involving few signals; synthesizing the whole synchronization is the task of the compiler.

- SIGNAL is its *own proof system:* desired properties can be expressed as (possibly non deterministic) SIGNAL programs, and processed by the compiler as additional equations. Checking for contradictions in the resulting program is the mechanism for proofs.
- The behavior of a program `P` in a context `C` may be easily studied as a program `C | P` (proofs, simulation...).

- The conditional graph associated with control equations is the universal tool for proving, distributing, optimizing SIGNAL programs.

To summarize, various services such as proof, compilation, distributed implementation, are all supported by the SIGNAL formal system. This releases the user from handling different formalisms and associated tools for these tasks.

SIGNAL is currently available under two different versions that were developed with different objectives. The INRIA H2 SIGNAL system provides the interface used in this article, and produces the intermediate level hierarchical code we have discussed. Sequential FORTRAN or C code is currently produced. Developments on distributed implementation are in progress based on this version. Tools for proving dynamical properties will be integrated in a short time.

The CNET-TNI V3 version is commercially available. A multiple windowing system of Macintosh style is provided for both program editing and on-line monitoring and supervision of the execution. Sequential C code is produced. Experiments have been performed based on this version to produce distributed OCCAM [16] code for a multi-Transputer system.

The SIGNAL environment has been experimented on significant applications in the area of signal processing and control: a speech recognition system, a radar system, a digital watch, a rail road crossing were the major ones.

Finally, the SYNDEX system [7] has been developed at INRIA to distribute automatically SIGNAL programs onto multiprocessor architectures; it uses the hierarchical conditional graph as input.

Appendix: a sample work of the compiler

Let us consider an excerpt of the `MOUSE` process presented in section 2.2.4, namely the `SIMPLE_MOUSE` process in which we specify also the subprocess `IN_INTERVAL`; moreover, we add the constraint (which is verified in the overall `MOUSE` process) that `START`s are also `CLICK`s:

```
START ^< CLICK
```

The `SIMPLE_MOUSE` process is the following:

```
process SIMPLE_MOUSE = { ? event START, CLICK, RELAX
                         ! event SINGLE, DOUBLE }
   (| START ^< CLICK
    | DOUBLE_CLICK := ((not START) default IN_INTERVAL {CLICK, START, RELAX})
                      cell RELAX
    | SINGLE := RELAX when (not DOUBLE_CLICK)
    | DOUBLE := RELAX when DOUBLE_CLICK
    |)
   where logical DOUBLE_CLICK
   process IN_INTERVAL = { ? X; event S, T
                           ! Y }
      (| BELONGS_TO_INTERVAL ^= (S default T default (event X))
       | (| WILL_BELONG := (not T) default S default BELONGS_TO_INTERVAL
          | BELONGS_TO_INTERVAL := WILL_BELONG $1 |)
       | Y := X when BELONGS_TO_INTERVAL
       |)
      where logical WILL_BELONG, BELONGS_TO_INTERVAL init false
   end
end
```

Its solved process, as calculated by the compiler, is as follows:

```
process SIMPLE_MOUSE_TRA = { ? event START, CLICK, RELAX
                             ! event SINGLE, DOUBLE }
   (| (| START ^= START |)
    | (| CLICK ^= (START default CLICK) |)
    | (| RELAX ^= RELAX |)
    | (| H_12_H := START default RELAX |)
    | (| H_15_H := CLICK default H_12_H
       | H_15_H() |)
    | (| SINGLE := RELAX when H_28_H |)
    | (| DOUBLE := RELAX when H_27_H |)
    | (| Y := CLICK when H_21_H |)
    | (| H_25_H := START default Y |)
    | (| H_26_H := RELAX default H_25_H
       | H_26_H() |)
    | (| H_14_H := when ((not H_12_H) default CLICK) |)
    | (| H_18_H := when ((not RELAX) default START) |)
    | (| H_24_H := when ((not START) default Y) |)
    |)
   where
   process H_15_H = { ? event H_15_H, H_14_H, H_18_H, RELAX
                      ! event H_21_H }
      (| H_15_H ^= WILL_BELONG ^= BELONGS_TO_INTERVAL
       | (| H_21_H := when BELONGS_TO_INTERVAL |)
       | (| BELONGS_TO_INTERVAL := WILL_BELONG $1
          | WILL_BELONG := (not RELAX) default H_18_H
                           default (BELONGS_TO_INTERVAL when H_14_H) |)
       |)
      where logical WILL_BELONG, BELONGS_TO_INTERVAL init false
   end;
```

```
   process H_26_H = { ? event H_26_H, H_24_H, START
                      ! event H_27_H, H_28_H }
      (| H_26_H ^= DOUBLE_CLICK
       | (| H_27_H := when DOUBLE_CLICK
          | H_28_H := when (not DOUBLE_CLICK) |)
       | (| DOUBLE_CLICK := ((not START) default H_24_H) cell H_26_H |)
       |)
      where logical DOUBLE_CLICK
   end
end
```

The hierarchy is represented as the embedding of declared subprocesses. If a clock is an external event, its name is the name of this external signal, otherwise it is named `H_i_H`.

For each clock named `X`, the solved process contains:

- its definition (for instance, `H_12_H := START default RELAX`) or constraint (`CLICK ^= (START default CLICK)`);
- a process with the same name containing the graph and clocks depending on `X` (see the processes `H_15_H` and `H_26_H`), or directly the subgraph of *synchronous* calculations (cf. the body of declared subprocesses).

Let us comment the `SIMPLE_MOUSE_TRA` process. In the hierarchy,

- events `START` and `RELAX` are *free* clocks; it is the reason why they appear at the top of `SIMPLE_MOUSE_TRA` with the constraint `X ^= X`;
- `CLICK` is constrained to be greater than `START` and thus is also placed at the top level (it would also be possible to consider that `CLICK` is free and `START` constrained);
- `H_12_H` and `H_15_H` are clocks built on more than one of those free clocks and then also appear at the top of `SIMPLE_MOUSE_TRA` with their definition;
- `H_15_H` is the clock of the boolean signals `WILL_BELONG` and `BELONGS_TO_INTERVAL`, it is used to build the clock `H_21_H` defined by the *true* values of `BELONGS_TO_INTERVAL`: `H_21_H` is under `H_15_H`; its definition and that of the signals `BELONGS_TO_INTERVAL` and `WILL_BELONG` are contained in the subprocess `H_15_H`;
- `RELAX`, `H_18_H` and `H_14_H` are "computation clocks" of `WILL_BELONG`; computation clocks associated with a given signal are exclusive clocks, i.e., clocks which do not have common instants (for instance, `H_18_H` is the "complementary" of `RELAX` in `START`); the expressions of definition of the signals (for example, `WILL_BELONG := (not RELAX) default H_18_H default (BELONGS_TO_INTERVAL when H_14_H)`) provide as a byproduct the conditional data dependencies;
- `SINGLE` (for instance) is built on `RELAX`, which appears at the top level, and `H_28_H`, which is under `H_26_H`, and thus it also appears at the top level (see also `DOUBLE`, `Y`, `H_25_H` and `H_26_H`); finally, the computation clocks `H_14_H`, `H_18_H` and `H_32_H` also appear at the top level.

The compiler does not synthesize a unique master clock for the SIMPLE_MOUSE process: no synchronization requirement is specified on the inputs START, CLICK and RELAX. This process is used as a subprocess of the MOUSE process. It can also be directly executed. We have then to define a communication protocol with its asynchronous environment. A scanning mode of asynchronous execution is described in the following process (to simplify the presentation, we consider that the process SIMPLE_MOUSE delivers the signal DOUBLE_CLICK as output):

```
process S_SIMPLE_MOUSE = { ? logical S_CLICK, S_RELAX, S_START
                           ! logical DOUBLE_CLICK }
   (| (| S_CLICK ^= S_RELAX
       | CLICK := when S_CLICK
       | RELAX := when S_RELAX
       | (| S_START ^= CLICK
          | START := when S_START |)
       |)
    | SIMPLE_MOUSE()
    |)
end
```

Here, the compiler synthesizes a single master clock: this process can be run in a master mode. The solved process is the following (we have kept only the skeleton of the program, dropping the definitions of the signals and the clocks which are only computation ones):

```
process S_SIMPLE_MOUSE_TRA = { ? logical S_START, S_CLICK, S_RELAX
                               ! event SINGLE, DOUBLE }
   (| (| H_6_H := event S_CLICK
       | H_6_H ^= S_RELAX
       | H_6_H() |)
    |)
   where
   process H_6_H = { ? event H_6_H; logical S_START, S_CLICK, S_RELAX
                     ! event SINGLE, DOUBLE }
      (| (| CLICK := when S_CLICK
          | CLICK ^= S_START
          | CLICK() |)
       | (| RELAX := when S_RELAX |)
       | (| H_33_H := CLICK default RELAX
          | H_33_H() |)
       | (| Y := CLICK when H_27_H |)
       | (| H_36_H := RELAX default START |)
       | (| H_37_H := Y default H_36_H
          | H_37_H ^= DOUBLE_CLICK |)
       |)
      where
      process CLICK = { ? event CLICK; logical S_START
                        ! event START }
         (| (| START := when S_START |)
          |)
      end;
```

```
      process H_33_H = { ? event H_33_H
                         ! event H_27_H }
         (| H_33_H ^= WILL_BELONG ^= BELONGS_TO_INTERVAL
          | (| H_27_H := when BELONGS_TO_INTERVAL |)
          |)
      end
   end
end
```

The clock H_6_H (which is the clock of the signals S_CLICK and S_RELAX) is the single root of the hierarchy; the clocks CLICK (which is the clock of the signal S_START), RELAX, H_33_H (which is the clock of the signals WILL_BELONG and BELONGS_TO_INTERVAL), Y, H_36_H, and H_37_H (which is the clock of the signal DOUBLE_CLICK) lie under H_6_H; the clock START lies under CLICK; the clock H_27_H lies under H_33_H.

As an example of sequential code generation, the C code generated from this simplified program is a loop

```
while(cs_simple_mouse());
```

with this function defined as follows:

```
extern logical cs_simple_mouse()
{
    h_6_h = TRUE;
    rs_click(&s_click,&h_4_h);
    if (!h_4_h) return FALSE;
    rs_relax(&s_relax,&h_4_h);
    if (!h_4_h) return FALSE;
    start = FALSE;
    h_33_h = s_click || s_relax;
    h_27_h = FALSE;
    if (s_click)
        {
        rs_start(&s_start,&h_4_h);
        if (!h_4_h) return FALSE;
        start = s_start;
        }
    if (h_33_h)
        {
        if (s_relax) will_belong = FALSE;
        else if (start) will_belong = TRUE;
        else will_belong = belongs_to_interval;
        h_27_h = belongs_to_interval;
        belongs_to_interval = will_belong;
        }
    y = s_click && h_27_h;
    h_37_h = y || s_relax || start;
    if (h_37_h)
        {
        if (start) double_click = FALSE;
        else if (y) double_click = TRUE;
        wdouble_click(double_click);
```

```
        }
    return TRUE;
}
```

The variable `belongs_to_interval` is initialized with `FALSE` and `rs_click`, `rs_relax`, `rs_start`, `wdouble_click` are input-output functions (the condition `(!h_4_h)` tests for the end of each input).

References

[1] A. Benveniste, G. Berry, "Real-Time systems design and programming", see this special section.

[2] A. Benveniste, P. Le Guernic, Y. Sorel, M. Sorine, "A denotational theory of synchronous communicating systems", INRIA Research Report 685, Rennes, France, 1987, to appear in *Information and Computation.*

[3] A. Benveniste, P. Le Guernic, "Hybrid Dynamical Systems Theory and the Signal Language", *IEEE transactions on Automatic Control*, 35(5), May 1990, pp. 535–546.

[4] A. Benveniste, P. Le Guernic, C. Jacquemot, *Synchronous programming with events and relations: the* Signal *language and its semantics,* IRISA Research Report 459, Rennes, France, 1989.

[5] P. Bournai, V. Kerskaven, P. Le Guernic, "Un environnement graphique pour la conception d'applications temps réel", *Colloque sur l'ingénierie des interfaces homme-machine*, Sophia-Antipolis, France, 1989, pp. 181–190.

[6] E. M. Clarke, E. A. Emerson, A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems*, 8(2), April 1986, pp. 244–263.

[7] N. Ghezal, S. Matiatos, P. Piovesan, Y. Sorel, M. Sorine, Syndex *Un environnement de programmation pour multi-processeur de traitement du signal. Mécanismes de communication,* INRIA Research Report 1236, Rocquencourt, France, 1990.

[8] N. Halbwachs, D. Pilaud, F. Ouabdesselam, A.-C. Glory, "Specifying, Programming and Verifying Real-Time Systems Using a Synchronous Declarative Language", in *Automatic Verification Methods for Finite State Systems* (Sifakis, ed.), Lecture Notes in Computer Science, Vol. 407, Springer-Verlag, Berlin, 1989, pp. 213–231.

[9] M. Le Borgne, A. Benveniste, P. Le Guernic, "Polynomial Ideal Theory Methods in Discrete Event, and Hybrid Dynamical Systems", in *Proceedings of the 28th IEEE Conference on Decision and Control*, IEEE Control Systems Society, Volume 3 of 3, 1989, pp. 2695–2700.

[10] B. Le Goff, "Inférence de contrôle hiérarchique : application au temps-réel", PhD thesis, Université de Rennes I, France, 1989.

[11] P. Le Guernic, T. Gautier, *Data-flow to von Neumann: the* Signal *approach,* INRIA Research Report 1229, Rennes, France, 1990, also in *Advanced topics in data-flow computing* (Gaudiot and Bic, eds.), Prentice-Hall, 1991, pp. 413–438.

[12] C. Le Maire, *Le langage* Signal *: un exemple en segmentation automatique de la parole continue*, INRIA Research Report 1217, Rennes, France, 1990.

[13] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends", in *Current Trends in Concurrency* (de Bakker and al., eds.), Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, Berlin, 1986, pp. 510–584.

[14] V. Roy, R. de Simone, *An* Autograph *Primer,* INRIA Technical Report, Sophia-Antipolis, France, 1989.

[15] D. Vergamini, *Verification by Means of Observational Equivalence on Automata,* INRIA Research Report 501, Sophia-Antipolis, France, 1986.

[16] Inmos Ltd, *The* Occam *programming manual,* Prentice Hall, 1984.

[17] "CSML", see this special section.

[18] "Esterel", see this special section.

[19] "Lustre", see this special section.

[20] "Statecharts", see this special section.