SIGNAL : a declarative language for synchronous programming of real-time systems

Thierry Gautier, Paul Le Guernic, Loïc Besnard

▶ **To cite this version:**

Thierry Gautier, Paul Le Guernic, Loïc Besnard. SIGNAL : a declarative language for synchronous programming of real-time systems. [Research Report] RR-0761, INRIA. 1987. inria-00075791

HAL Id: inria-00075791
https://inria.hal.science/inria-00075791

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 761

SIGNAL : A DECLARATIVE LANGUAGE FOR SYNCHRONOUS PROGRAMMING OF REAL-TIME SYSTEMS

Thierry GAUTIER
Paul LE GUERNIC
Loïc BESNARD

NOVEMBRE 1987

SIGNAL : A DECLARATIVE LANGUAGE FOR SYNCHRONOUS PROGRAMMING OF REAL-TIME SYSTEMS

Thierry GAUTIER, Paul LE GUERNIC
IRISA / INRIA

Loïc BESNARD
CICB / CNRS

Campus de Beaulieu, 35042 Rennes Cédex, FRANCE
Publication Interne n° 379 - Novembre 87 - 22 pages.

ABSTRACT

We present an applicative language, SIGNAL, designed to program real-time systems. The language is based on a synchronous *notion of time. We assume the execution of operations to have a zero logical time duration ; then, the sequence of communication events determines entirely a temporal reference. The ordering of the runable operations is limited only by the dependencies between the calculi : this is the point of view of data flow languages. SIGNAL is a* data flow language *(where the potential parallelism is implicit), which permits a structural description of interconnected processes. SIGNAL handles possibly infinite sequences of values (called signals) characterized by an implicit clock which specifies the relative instants (with respect to other signals) at which these values are available. Specific operators, such as delay, undersampling, deterministic merge, are designed to express temporal relations between different signals : in this way, a SIGNAL program expresses both functional and temporal relationships between all the involved signals. The language is semantically sound, and its declarative style allows to derive, by a simple projection on the commutative field* Z/3Z, *a complete static calculus of the timing of any SIGNAL process, called its* clock calculus. *Hence, the language SIGNAL is also a formal system to reason about timing and concurrency. The clock calculus is completed together with the dependency analysis of a given program. This leads to a* conditional dependence graph *in which the edges may be labelled by the involved clocks. From this graph, we generate code for a sequential machine, but it appears to be the suitable level to study the implementation on a multiprocessor architecture.*

This paper has been published in the proceedings of the third Conference on Functional Programming Languages and Computer Architecture (G. Kahn, editor, Lect. Notes in Computer Science, **274**, *Springer Verlag).*

PAPIER RECUPERE ET RECYCLE

SIGNAL : UN LANGAGE DECLARATIF POUR LA PROGRAMMATION SYNCHRONE DE SYSTEMES A TEMPS REEL

RESUME

On présente un langage applicatif, SIGNAL, conçu pour la programmation de systèmes à temps réel. Le langage est basé sur une notion de temps synchrone. *On suppose que l'exécution de chaque opération a une durée logique nulle ; la suite des événements de communication détermine alors complètement une référence temporelle. L'ordonnancement des opérations exécutables n'est contraint que par les dépendances existant entre les calculs : ceci est le point de vue des langages à flots de données. SIGNAL est un* langage à flots de données *(le parallélisme potentiel est implicite) qui permet une description structurelle de processus interconnectés. Les objets de base du langage sont des suites éventuellement infinies de valeurs (appelées signaux) caractérisées par une horloge implicite qui spécifie les instants relatifs (par rapport aux autres signaux) auxquels ces valeurs sont disponibles. Des opérateurs spécifiques comme le retard, le sous-échantillonnage, le mélange déterministe, permettent d'exprimer les relations temporelles entre plusieurs signaux : un programme SIGNAL exprime ainsi tout à la fois les relations fonctionnelles et les relations temporelles existant entre tous les signaux mis en œuvre. Le langage est fondé sémantiquement ; en outre, son style déclaratif induit de manière très simple (par une projection sur le corps commutatif* Z/3Z*) un calcul statique complet, appelé* calcul d'horloges, *des synchronisations de tout processus SIGNAL. Le calcul d'horloges est effectué en parallèle avec l'analyse des dépendances d'un programme donné. Cela conduit à la construction d'un* graphe de dépendances conditionnées *dans lequel les arcs peuvent être étiquetés par les horloges concernées. Ce graphe est le point de départ pour la production de code, en particulier il apparaît être le bon niveau pour l'étude de l'implantation sur une architecture de multi-processeur.*

1 Introduction

Real-time programming has almost been traditionally monopolized by imperative asynchronous languages such as ADA [1], LTR [2], OCCAM [3], etc. These languages involve high level concepts – for example, the rendez-vous mechanism, well studied in the CSP formalism [4] – to deal with parallelism and communication, and they generally possess some specific primitives – for example, a delay statement – to handle the time. Unfortunately, the problem is that these languages are known to be essentially nondeterministic, that is, one is in fact completely unable to specify the effective duration of any statement and, by the way, to know exactly the time at which a given process executes a given action. Due to the asynchronous character of the high level communication features, it is impossible to fully abstract implementation considerations.

However, for some years, a new programming methodology has emerged, specially devoted to real-time systems. The main idea supporting this new style is to properly delimit the inherent asynchronous points in a given application and to program the whole remaining part of the application – which is in fact the heart of the problem in a high level point of view – in a purely *synchronous* way. Several programming languages, known as *synchronous languages*, have been designed for this purpose. The two pioneering languages are the imperative language ESTEREL [5] and the applicative data flow oriented language SIGNAL [6,7]. Here we present and justify the declarative style of SIGNAL which permits a programmation very close to the specification (either mathematical or graphical) of a problem and leads to an elegant synchronization formal calculus. LUSTRE [8] is an other synchronous language, coming from Lucid [9]. A more precise classification of these languages is suggested in [10].

Now, what are the basic principles of *synchronous systems* ? For us, the essential characteristic of a *real-time* system is that one can always bound the response times of the system [11]. As a consequence of this definition, it must be possible to statically (i.e., independent of any input value) bound the *number* of operations between two external events on one hand, and the memory size necessary to the execution on the other hand. Under these conditions, let us project the time along two dimensions : the first one, that we name *logical time* dimension, will permit to have a first formal timing calculus ; the second one is the physical time dimension. Then we can assume the following hypothesis : the execution of any operation has a *zero logical time duration*. Consequently, *a temporal reference is entirely determined by the sequence of* all *communication events* (and not only by the input events as this is the case in ESTEREL or LUSTRE : we shall see that one can define in SIGNAL output data which are "more frequent" than input ones). Consider, as a simple example, a real-time system which has two input *ports* : one, named *x*, carries successive values x_1, x_2, etc. and other, an interrupt port, named *s*. Then, according to

the synchronous point of view, the specification of an history [12] of the system has the following form :

$$x_1, (x_2, s_2), x_3, s_4, \ldots$$

where both the values and their global ordering are specified. The integer index $t = 1, 2, 3, 4, \ldots$ defining successive "instants" is then considered as the proper notion of time for the synchronous system. Moreover, contrary to classical real-time languages which include only one notion of universal absolute time (this is clearly not sufficient when we consider a time unit as being just a repetitive event), we have a multiform notion of relative time. For a given task, the time is just the global ordering of all the data of the task. This is a local timing preserving the modularity of a system since the cooperation between several sub-tasks defines a new temporal reference for the global system. Note that the essentially nondeterministic character of the asynchronous communications with external world is concentrated in the mechanism which decides the global ordering and is not propagated inside the body of the synchronous system. *Reactive systems* [13], which have been defined to describe complex systems interacting with their environment are privileged candidates for synchronous programming.

As a consequence of the logical instantaneity of actions, as soon as its data are available, a given operation is instantaneously performed. The availability of these data, and so the ordering of the runable actions in a task, is determined by the dependencies between the data and operations in the task. This is exactly the point of view of data flow languages in which the potential parallelism is just limited by the data dependencies. Although usual data flow languages are asynchronous, it is easy to see that data flow concepts are totally compatible with the synchronous principles [14]. In data flow languages, the potential concurrency is expressed through the use of a functional programming style.

SIGNAL is a synchronous data flow oriented language. Its new notion of time allows to implement SIGNAL programs without any overhead, even on classical architectures.
The rest of this paper is organized as follows : in the second section, we define the operators dealing with time in SIGNAL ; in the third section, we present the notion of modularity in the language and detail some examples ; then the fourth section gives the basis of the formal synchronization calculus and briefly introduces a few implementation considerations.

2 The time

The basic objects of the language are named *signals*. A signal v is characterized by the ordered sequence (v_t) of its (typed) values, and by its *clock*. The clock specifies the *relative* instants at which the values of the signal are available (contrary to variables in classical languages, these

values are not persistent : they are available only at these logical instants). These instants are not absolute ones : clocks are just the good way to express *temporal relationships* between different signals. In the SIGNAL language, these temporal relations are in fact a byproduct of the operators which are used to handle signals and thus the clock of a given signal can be implicit.

2.1 Elementary processes

A SIGNAL process describes both functional and temporal relationships between signals. For example, the signal *v* given by

$$\forall t \quad v_t = zv_t + 1 \tag{2-1}$$

is obtained in SIGNAL by the equation

$$v := zv + 1 \tag{2-2}$$

(where the *t* index has been dropped). This elementary process defines the (output) signal *v* from the (input) signal *zv* (fig. 1). Like the other classical operators, the addition is considered as an instantaneous function. So, *v* and *zv* are available at the same instants, and, at each one of these instants, the equation (2-1) is valid. More generally, all the signals implied in an expression containing only instantaneous functions must *have the same clock* (they are also said to be *synchronous*). Constant values, such as *1*, are available at any clock : here, *1* is available at the clock of both *v* and *zv*.

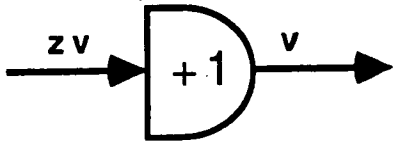

Figure 1.

2.2 The delay

Now if we want to give access not to the current value of a signal, but to its past value, we write

$$\forall t \quad zv_t = v_{t-1} .$$

This is immediately translated into SIGNAL using the *delay* operator : the process

$$zv := v \$ 1 \tag{2-3}$$

defines the output signal *zv* which is "one-step delayed" over the input signal *v* (fig. 2). An initial value

$$zv_0 = v0$$

(where *v0* denotes a constant) is specified by the declaration

$$zv \textit{ init } v0 .$$

Here again, *zv* and *v have the same clock* : they are available at the same instants, but at each one of these instants, *zv* possesses the value taken by *v* at the previous one.

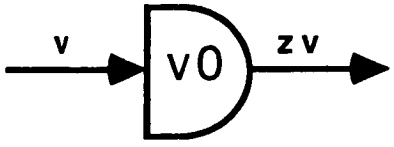

Figure 2.

2.3 Composition of processes

Given several processes such as (2-2) and (2-3), we can compose them together, thus defining a new process :

$$(\mid v := zv + 1 \mid zv := v \$1 \mid) . \qquad (2\text{–}4)$$

This is a general method when we have equations defining synchronous signals. For example, by introducing the right delays, the equation

$$v_t = v_{t-1} + 1$$

is transformed into the following system of equations

$$\{ v_t = zv_t + 1, \quad zv_t = v_{t-1} \}$$

which is immediately translated into the SIGNAL process (2–4), using the composition of processes. Obviously we can compose in the same way not just elementary processes but also processes already defined by a system of equations.
The composition identifies signals having the same names in different processes, thus connecting

corresponding output and input *ports* (or signals). We do not detail here the formal definition of the composition, but note that these connections are established only from the outputs of any process to the inputs having the same name in the *other* processes : in a given process, an input and an output port possessing the same name are not connected by the composition (they still denote distinct signals). The process (2–4) has two output signals *v* and *zv*, but has no input (fig. 3) : to avoid connecting one input port with several output ports (which would imply some nondeterministic merge), any connected input port becomes invisible for the outside. Moreover, a process cannot have several output ports with the same name, and the composition is defined if and only if there are no outputs with the same names in the composed processes – i.e. there are not several definitions for one signal. The composition is then associative and commutative (the order of the equations has no importance).

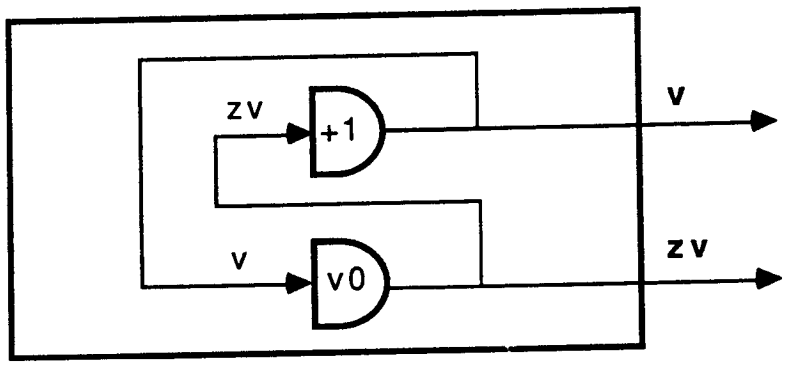

Figure 3.

If *zv* is initialized with 0, the process (2–4) clearly defines a counter of its own clock instants. Its successive values (at the output *v*) are 1, 2, 3, 4, 5, etc., but this process does not determine any clock for the signal *v* : this clock is defined by the context where the process is used.

2.4 The undersampling

Now, suppose that we want to count the occurrences of a given event, for example, the *true* occurrences of a boolean signal *c*. Then, if we define the signal *a* carrying only these *true* occurrences, *a* will be extracted from *c* and obviously less frequent than *c*. For this we have in SIGNAL the *undersampling* operator : the process

$$a := b \textit{ when } c$$

(where *c* is a boolean signal) delivers the input signal *b* at the output *a* when both *b* and *c* are available and when *c* is *true*. The input signals *b* and *c* may have different clocks ; in an obvious order relation on clocks, we can just say that the clock of *a* is smaller than both clock of *b* and

clock of *c*. We shall see later the right formalism to reason about clocks.
Note that the SIGNAL process

```
c := event x
```

delivers always *true* boolean signal *c* which is available exactly when the signal *x* is available : it may be considered as the proper clock of the signal *x*. By this way, the process

```
a := b when event x
```

delivers *b* at the output *a* when both *b* and *x* inputs are available.
Let us come back to our previous problem : the signal *a* carrying only the *true* occurrences of a boolean signal *c* is specified by the process

```
a := true when c .                                    (2-5)
```

2.5 Explicit synchronization

In order to count the occurrences of the signal *a* specified in the process (2-5), it is sufficient to deliver the signal *v* defined in the process (2-4) at the proper clock of *a* : this is realized through the use of an explicit synchronization. Given the input signals *a* and *v*, the process

```
synchro a, v
```

(which has no outputs) asserts that *a* and *v* must have the same clock.
Thus, the signal *v* counting the *true* occurrences of a boolean signal *c* is defined by the process

```
(| a := true when c |
   synchro a, v |
   v := zv + 1 |
   zv := v $1 |) .
```

2.6 The merge

Now, let us try to define a counter *modulo n* (where *n* is any integer value) : it will be necessary to realize a (deterministic) merge of two signals (typically, one of them will be a reset signal and the other one an accumulator). This is made possible in SIGNAL through the use of the *merge* operator. The process

```
v := a default b
```

delivers at the output *v* the deterministic merge (obtained by giving a priority to *a*) of the input signals *a* and *b* : when *a* is available, its value is delivered on *v* ; when it is not, the value of *b* (when *b* has one) is delivered on *v*. Input signals *a* and *b* may have distinct clocks, but here, the signal *v* is more frequent than *a* and *b* and its clock is greater than those of *a* and *b*. This operator is very important since it permits to define signals which are more frequent than the inputs of a system. For example, we can generate instants "between" the instants of an input signal ("time multiplexing" : see [10] for details).

Then, the signal *v* defining a counter modulo *n* is reset each time its previous value is equal to *(n – 1)*. Supposing this counter is supplied at the clock of some signal *a*, it is defined by the following process :

$$
\begin{array}{c}
(\mid \mathrm{zv} := \mathrm{v}\ \$1 \mid \\
\mathrm{v} := (0\ \textit{when}\ \mathrm{zv} = \mathrm{n}-1)\ \textit{default}\ (\mathrm{zv} + 1) \mid \\
\textit{synchro}\ \mathrm{a}, \mathrm{v}\ \mid) .
\end{array}
$$

We have presented the main operators available in SIGNAL : these are instantaneous functions, delay, undersampling, and merge ; the *event* operator gives access to the clock of a signal and the *synchro* process allows explicit synchronization. Another useful operator is derived from the previous ones : since the values of the signals are not persistent in the language, it may be necessary to memorize a value during several instants of a given clock. In this way, the process

$$\mathrm{a} := \mathrm{b}\ \textit{cell}\ \mathrm{c}$$

(where *c* is a boolean signal) delivers at the output *a* the last available value of *b* when either *b* or *c* is available and *c* is *true* (remember that *c* may be any clock if we use the expression *event* x).

3 The modularity ; some examples

To illustrate the potential modularity in the language, we would like to define a *model* of process.

3.1 Model of process

Let us come back to our example of counter modulo and imagine two different cases : the first one is the classical counter which is reset whenever it exceeds some value ; in the second one, we have an additional external signal which constrains extra resets.

First, to obtain a counter modulo of any integer value, we use a formal parameter, say *v0*. Parameters represent constant values and are not characterized by a given clock : their value are available at the clock determined by the context in which they are used. Then, we define the

following process, named *topmod*, that we shall be able to use as well in the two different cases we have distinguished :

```
topmod { integer v0
         ? logical hreset, iev
         ! integer v ; logical oev }
= (| synchro iev, zv |
     zv := v $1 |
     v := (0 when event hreset) default (zv + 1) |
     oev := true when zv >= v0
   |) / zv
where integer zv init v0
end
```

This process defines two output signals : the counter itself (*v*) and a signal *oev* which clearly represents the reset instants of a simple counter modulo. It takes two input signals : *hreset* is supposed to specify all the reset instants and *iev* is a synchronization signal which assigns a given clock to the counter.

Notice that the operation

$$P / a_1, ..., a_n$$

denotes the *masking* of the *output* signals a_1, ..., a_n in the process *P* : the masked signals may be considered as local signals for the resulting process, which, from the outside, can be seen as a "black-box". Here, the signal *zv* is local for the process *topmod*.

A SIGNAL process is fundamentally "polychronous" : generally it deals with several signals which do not have the same clock. Let us analyse the synchronizations in the process *topmod*. We call here *vz* the local signal (0 *when event* hreset) and *vp* the signal (zv + 1). From the definitions of the operators, we obtain the following considerations immediately : on one hand, *v*, *zv*, *vp* and *iev* are synchronous and on the other hand, *vz* and *hreset* are synchronous as well ; moreover, *v* is more frequent than *vz*. So, the output *v* is more frequent than the input *hreset*, and we derive the constraint that the input *iev* is necessarily more frequent than the input *hreset*. The process is runable in a context where this constraint is respected (in particular, there is no implicit clock associated to a given process). Notice that without the *synchro* specification, the counter would be nondeterministic, in the sense that its clock would not be constrained (but a constraint can be fixed by the context of the process). Obviously, we would like to have a formalism to express these considerations : this is the subject of section 4.

3.2 Process instanciation

We are ready to instanciate a process model in a given context by reproducing this model. For example, a simple counter modulo *n* is defined by the process

topmod(n – 1) ! oev : hreset @ hreset

depicted on figure 4.

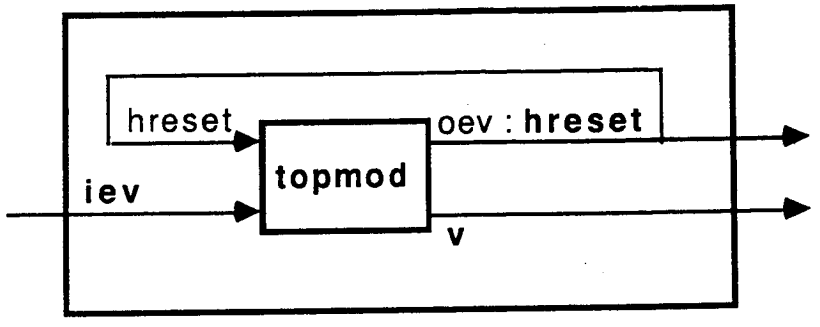

Figure 4.

The effective parameter *(n – 1)* corresponds to the formal parameter *v0*. In this instanciation of the process *topmod*, its output *oev* is renamed *hreset* and is looped on the input *hreset* : the reset instants are just those specified by the output *oev*. Note that this new process requires no more synchronization constraints.

Here we have introduced two other operations : *relabelling* and *connection*. Given any process *P*,

$$P\ ?\ a_1 : b_1, \ldots, a_n : b_n$$

and

$$P\ !\ a_1 : b_1, \ldots, a_n : b_n$$

respectively denote the processes obtained by renaming every input port (respectively, output port) a_i into b_i ;

$$P\ @\ a_1, \ldots, a_n$$

denotes the process where every output a_i of *P* is connected to the input port having the same name (remember that the composition does not realize this connection in a given process). Of course, do not confuse composition, masking, relabelling and connection which are just

interconnection operations acting on processes ("P-operators"), with the operators defining signals ("S-operators") ! The mechanism of establishing connections by handling explicitly the names of inputs and outputs is very close to the classical graphical description used in automatic (block diagrams). By the way, we intend to develop a graphical programming interface for the language SIGNAL. On the other hand, it is also possible to program without worrying too much about identical names : we have defined a more classical positional notation (not detailed here) where relabellings are made implicit.

Now, the counter with an external reset is specified by the process

(| hreset := hreset *default* oev |
topmod(n – 1) |) / hreset

depicted on figure 5.

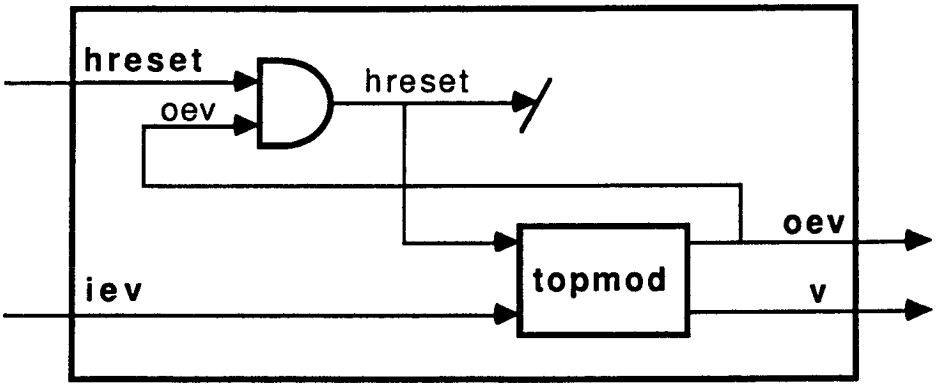

Figure 5.

The input signal *hreset* specifies the additional reset instants : it is merged with the output *oev* of *topmod* in order to obtain all the reset instants. Here again, *iev* has to be more frequent than *hreset*.

3.3 An other example

Now we give an example of utilization of the process *topmod* where the constraint is satisfied. This example is a simple *chronometer* which counts the seconds and the minutes, with an explicit reset : it can be clearly specified with two different instanciations of *topmod*, one to count the seconds and the other to count the minutes. The signal *iev* gives the clock of the outputs (i.e. the instants where the values of the chronometer are present). The program is :

```
chronometer { ? logical iev, reset
              ! integer seconds, minutes }
=
   (| (| sreset := (event reset) default (event oev) |
        topmod(59) ? hreset : sreset ! v : seconds |) / sreset |
      ievint := (event oev) default (event reset) |
      (| mreset := (event reset) default (event oev) |
        topmod(59) ? iev : ievint, hreset : mreset |
        minutes := v cell event iev |) / oev, v, mreset
   |) / oev, ievint
where
   logical ievint, sreset, mreset
process
   topmod { ... }
   = ...
   end
end
```

The seconds' counter is reinitialized when there is an external reset event or when the maximal value has been reached : this is realized through the signal *sreset*. The same remark is valid for the minutes' counter (signal *mreset*). The signal *ievint* gives the clock of the latter counter : the minutes' counter is activated when either the seconds' counter reaches its maximum or an external reset event occurs. The *cell* operator synchronizes the signal *minutes* with the signal *seconds* : this is required to display the values. Note that *topmod* is a local process for the *chronometer*.

4 Formal synchronization calculus

We have pointed out the crucial necessity to formally reason about time in SIGNAL programs : this permits to statically (i.e., at compile time) assign a clock to every signal (with respect to the other signals) and to bring out synchronization constraints. In particular, nondeterministic programs have to be detected, as well as incorrectly synchronized ones. Moreover, since it has to define the timing of any program, the formal synchronization calculus is a basic tool for the implementation. Thanks to the declarative style of the language, we shall see that the equations of this calculus are a direct projection of the SIGNAL program.

4.1 The clock calculus

One can observe that the only relevant informations to reason about clocks are concentrated in *three* values. These are : presence of a signal, absence of a signal, and values of boolean signals (we have seen that boolean conditions can introduce new clocks – in an undersampling for example : this explains why a simple boolean calculus is not sufficient). So we distinguish two types of signals : boolean ones and other signals. Values of boolean signals may be represented by the set { 0, 1, 2 } : 0 is the absence of value, 1 is the *true* value and 2 is the *false* value ; values of other signals are represented by the set { 0, 1 } : 0 is the absence of value and 1 the presence of a value. Then, we endow the set { 0, 1, 2 } with the structure of the commutative field *Z/3Z* and we can project every SIGNAL process on *Z/3Z*. In this field, since the map

$$x \to x^2$$

maps respectively 0 onto 0, 1 onto 1, and 2 (which is also written as -1) onto 1, any expression of non boolean signals may be represented by a function in x^2. Moreover, since

$$x^3 = x ,$$

any function on *Z/3Z* is polynomial of degree 2 at most. For each one of the S-operators we have presented, and for boolean (*not*, *or*, *and*) and non boolean functions, the following table gives the equations associated to the given process.

expressions	boolean result (values)	non boolean result (clocks)
y := *not* x	$y = -x$	$y^2 = x^2$
y := a *or* b	$y = ab\,(1 - (a + b + ab))$ $a^2 = b^2$	$y^2 = a^2 = b^2$
y := a *and* b	$y = ab\,(ab - (a + b + 1))$ $a^2 = b^2$	$y^2 = a^2 = b^2$
y := $f(a_1, ..., a_n)$	$y^2 = a_1^{\,2} = ... = a_n^{\,2}$	$y^2 = a_1^{\,2} = ... = a_n^{\,2}$
y := x $1	$y^2 = x^2$	$y^2 = x^2$
synchro $a_1, ..., a_n$	$a_1^{\,2} = ... = a_n^{\,2}$	$a_1^{\,2} = ... = a_n^{\,2}$
c := *event* a	$c = a^2$	
y := a *when* c	$y = a\,(- c - c^2)$	$y^2 = a^2\,(- c - c^2)$
y := a *default* b	$y = a + b - a^2b$	$y^2 = a^2 + b^2 - a^2b^2$

The equations of this table are easy to verify : for example, the map $x \to -x$ maps respectively 0, 1 and -1 onto 0, -1 and 1, which represents the boolean function *not*. The other boolean functions (*or*, *and*) may be verified in the same way. One can have an intuitive justification of the equations. For instance, non boolean functions result only here in synchronization of all signals : their clocks are equal and this is exactly expressed by the equality of the square values in *Z/3Z*. For the delay too, we only express the fact that both the output and the input have the same clock : $y^2 = x^2$. The *event* operator delivers the clock of a given input signal *a*, so $c = a^2$. The undersampling operator delivers *a* when *c* is available (expressed by c^2) and when *c* is not *false* (expressed by $-1-c$, which is 0 when $c = -1$) : $y = ac^2(-1-c) = a(-c-c^2)$; if we square this equation, we obtain the clock of the output : $y^2 = a^2(-c-c^2)$. For the merge y := a *default* b, the equation $y = a + b - a^2b = a + b(1 - a^2)$ (for boolean values) expresses that *y* takes the values of *a* when *a* is available (this is expressed by *a*) and the values of *b* when *a* is *not* available (this is expressed by $b(1 - a^2)$).

Every SIGNAL process is thus represented by a system of equations of degree 2 at most in *Z/3Z*. Then, the temporal analysis of a SIGNAL program is exactly the analysis of this system, what we call the *clock calculus* [7]. We do not detail here the principles of this analysis. Let us simply examine on some examples the clock synthesis we are able to do.

The following process is clearly incorrect since we add two signals whose clocks are mutually exclusive :

$$\begin{array}{c}(|\ x := a\ \textit{when}\ (a > 0)\ | \\ y := a\ \textit{when}\ (\textit{not}\ (a > 0))\ | \\ z := x + y\ |)\ .\end{array}$$

Its clock calculus is expressed through the following equations (where *c* represents the condition $a > 0$) :

$$\begin{array}{c}a^2 = c^2 \\ x^2 = a^2(-c-c^2) \\ y^2 = a^2(c-c^2) \\ z^2 = x^2 = y^2\ .\end{array}$$

Note that expressions such as $a > 0$ are considered as primitive boolean expressions.
Elementary substitutions yield the constraint

$$c = 0$$

which means that the process is blocked (since 0 is the absence of event).

More generally, if the system associated to a given process has a solution such that for every input or output, the associated variable in *Z/3Z* is not necessarily null, then the process is not blocked, on any of its ports.

The following process is again incorrect :

$$(|\ x := a \textit{ when } (a < b) \mid z := x + b\ |)\ .$$

Its clock calculus is :

$$\begin{aligned} a^2 &= b^2 = c^2 \\ x^2 &= a^2\,(-\,c - c^2) \\ z^2 &= x^2 = b^2 \end{aligned}$$

(*c* denotes here the condition $a < b$). We obtain the equality

$$c\,(c - 1) = 0$$

which expresses in fact the following constraint on the *values* of the inputs :

"(a and b are not defined) or $a < b$*".*

More generally, if the system has a solution such that the values of the input signals are free, then the process can be synchronized.

Now, we have the following clock calculus for the process *topmod* of section 3 (recall that *vz* and *vp* respectively denote the signals *(0 when event hreset)* and *(zv + 1)* ; *c* denotes the condition $zv \geqq v0$) :

$$\begin{aligned} iev^2 &= zv^2 = v^2 = vp^2 \\ vz^2 &= -\,(hreset^2) - hreset^2 = hreset^2 \\ v^2 &= vz^2 + vp^2 - vz^2\,vp^2 \\ oev^2 &= oev = -\,c - zv^2 \end{aligned}$$

which implies

$$v^2 = iev^2$$
$$oev = v^2 (- c - 1)$$
$$hreset^2 = iev^2\ hreset^2$$

This last equation illustrates again the fact that a SIGNAL process can exhibit synchronization constraints at its input ports. This is a fundamental remark, since the environment of this process must be in accordance with such constraints. Note that this process is fully time-correct. The equation $hreset^2 = iev^2\ hreset^2$ means that the signal *hreset* must be less frequent than the signal *iev*. For the process *topmod*, the different clocks are :

$$h_1 = \{\ hreset,\ vz\ \}$$
$$h_2 = \{\ iev,\ v,\ vp,\ zv\ \}$$
$$h_3 = \{\ oev\ \}$$

where $h_i = \{\ s_1, ..., s_i\ \}$ denotes that the signals $s_1, ..., s_i$ have the same clock h_i.

4.2 The conditional dependence graph

The clock equations express the timing of a given process. In order to produce the code of the process, it is also necessary to analyse the dependencies between the calculi of the process. For each instant, we must be able to generate control independently of the instants and also to detect cycles in the dependencies. As a matter of fact, the right way to do this is to construct, during the analysis of the process, its instantaneous *conditional dependence graph*. The nodes of the graph are the different signals and clocks involved in the process. Of course, the edges express the dependencies, but they are labelled here by clock expressions : this allows to accept cycles which are not real short circuits.

For the process *topmod* the conditional dependence graph is depicted on figure 6.

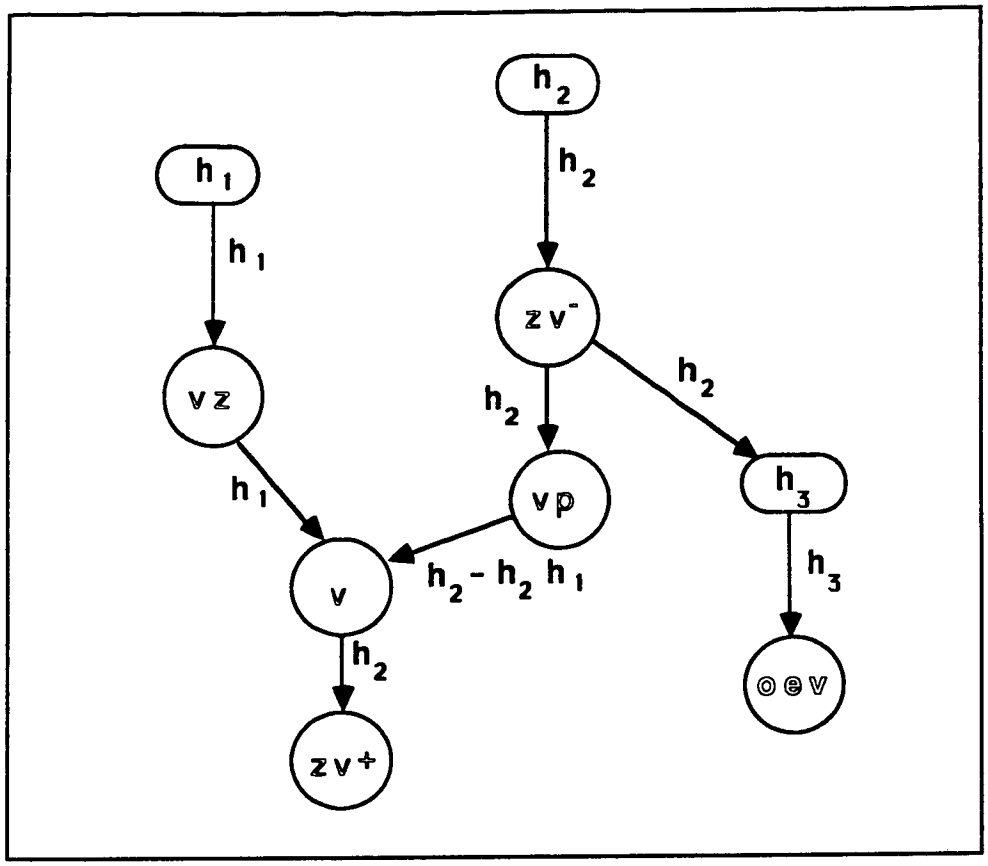

Figure 6.

On this example, one can note that two nodes, zv^- and zv^+, are associated to the signal *zv* defined by the delay operator : since the delay can be seen as a memory, two nodes are clearly required to have an instantaneous graph. In fact the graph is made from elementary conditional dependence graphs associated to the operators. For example, the values of *v* (obtained by a *default* operator) result from two calculi, one when *hreset* is present (clock h_1), and the other one when *vp* is present but *hreset* is not. Note that the values of the signal *vp* are defined at the clock h_2 and are available at the clock $h_2 - h_1 h_2$.

The conditional dependence graph allows us to accept some cycles, which are not short circuits. Consider the following program :

$$(|\ y := x \textit{ when event } b \mid u := y + z \mid z := u\ \$1 \mid x := a \textit{ default } u\ |)$$

The clock calculus gives :

$$y^2 = x^2 b^2$$

$$u^2 = y^2 = z^2$$
$$x^2 = a^2 + u^2 - a^2 u^2 .$$

Simple substitutions yield :

$$x^2 (1 - b^2 + b^2 a^2) - a^2 = 0$$

Both

$$\text{(i)} \quad x^2 = a^2 + b^2 - a^2 b^2$$

and

$$\text{(ii)} \quad x^2 = a^2$$

express solutions for this equation.

The conditional dependence graph of this example is depicted on figure 7.

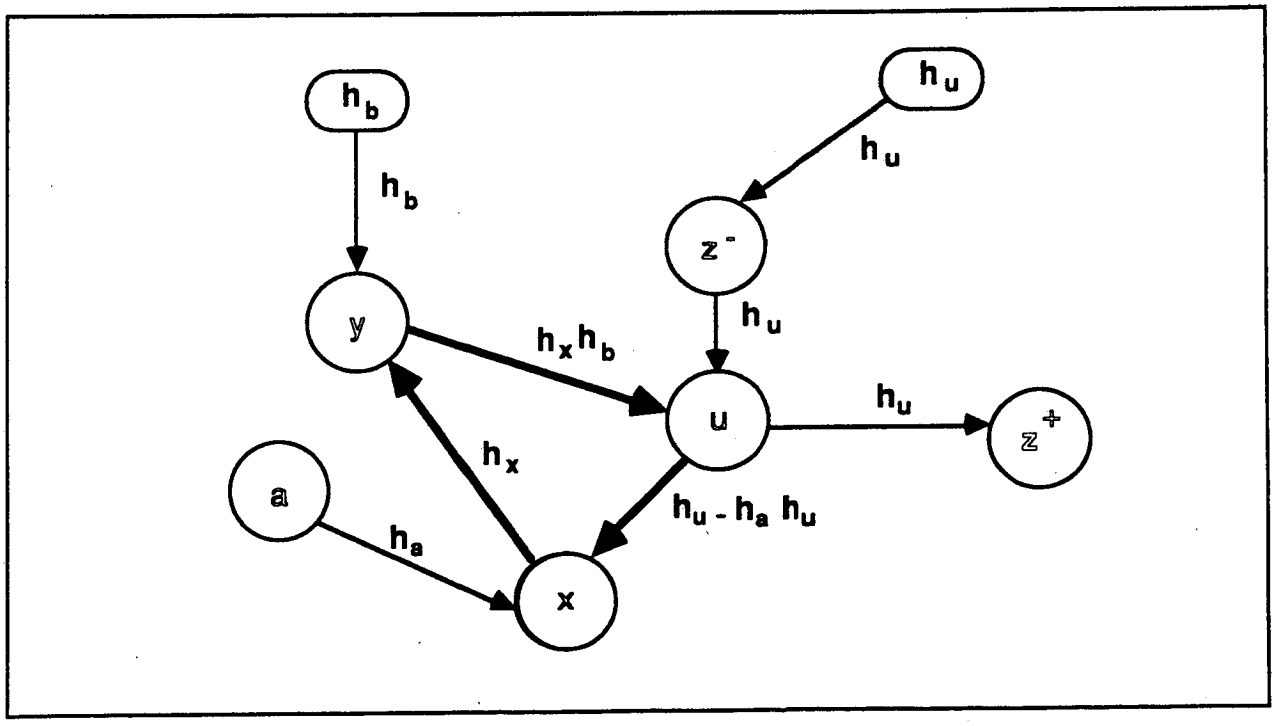

Figure 7.

There are several correct timings for this program. The more frequent timing for *x* is the

supremum of the clocks of *a* and *b* (i), whereas the less frequent one is the clock of *a* (ii). If the clock of *a* is chosen, this means that the value of *u* is never chosen to compute *x*, so that *there is no short circuit* in this case. But, if the most frequent one is chosen, then *u* will be sometimes used to compute *x*, which causes a short circuit.

The general method to detect short circuits is the following :
When a cycle is detected, the product of the clocks expressions which label the edges of the cycle is done. If this product is not null then the cycle is a short circuit, otherwise the graph is said *circuit-free* and the process is *computable*. The demonstration of this theorem is in [7]. Intuitively, a null product means that the signals involved in the cycle will never be simultaneously available.

When the process is fully time-correct and without any short circuit, the conditional dependence graph together with the solved clock calculus are the basic tools to generate code. The clock calculus allows us to produce a control automaton which governs a calculi automaton obtained from the dependence graph. At the present time, the compiler generates in this way sequential code in order to have a functional simulation of the algorithms.

5 Conclusion

We have presented the synchronous data flow oriented language SIGNAL. This language has already been used in different domains. For example, we have programmed in SIGNAL a complete digital watch with chronometer, alarm, etc. A privileged domain for the language is the field of signal processing algorithms : as particular applications, we have written and simulated a modem, an algorithm for the detection of abrupt changes in spectral characteristics of digital signals, etc. Let us mention as well an attempt to describe in SIGNAL the internal architecture of a microprocessor.

The clock calculus allows us to *statically* (i.e. at compile-time) verify the temporal correctness of the processes. In fact, it can be seen as a formal system to reason about timing. The first version of SIGNAL implements a restricted boolean clock calculus. The implementation of the calculus we have explained here is in preparation, it uses elementary elimination techniques to simplify the equations. Moreover, it will be extended into some *dynamical clock calculus* in order to fully handle the delays. The conditional dependence graph, together with this clock calculus, is a key tool to study concurrency and appear to be the suitable level to achieve a parallel implementation. A fundamental problem is to allocate the various calculi to different processors. For this, a first target is to reduce the number of nodes of the graph we have to distribute. We currently study different criterions to obtain a partition of this graph : a natural one is to partition according to the clock of the signals. Then, other partitions, such that every input in any sub-graph must precede

every output, result in what is called the "granules" [15], which may be considered as non interruptable units. In this way, we are going towards a possibly aided implementation of SIGNAL programs onto a multiprocessor architecture.

REFERENCES

[1] *Reference Manual for the ADA Programming Language* ; U.S. Department of Defense, 1983.

[2] *Manuel officiel de référence LTR 3 (indice 2)* ; CELAR, Bruz, July 1985.

[3] *OCCAM Programming Manual* ; INMOS Limited, 1983.

[4] C.A.R. HOARE : *Communicating Sequential Processes* ; Comm. of the ACM 21 (8), August 1978, 666 – 677.

[5] G. BERRY, L. COSSERAT : *The ESTEREL Synchronous Programming Language and its Mathematical Semantics* ; Seminar on Concurrency, July 1984, S.D. Brookes, A.W. Roscoe and G. Winskel, editors, Lect. Notes in Computer Science, 197, Springer Verlag, 1985, 389 – 448.

[6] P. LE GUERNIC, A. BENVENISTE, P. BOURNAI, T. GAUTIER : *SIGNAL : a data flow oriented language for signal processing* ; INRIA, Rennes, Research Report n° 378, March 1985.

[7] P. LE GUERNIC, A. BENVENISTE : *Real-Time, Synchronous, Data-Flow Programming : the Language SIGNAL and its Mathematical Semantics* ; INRIA, Rennes, Research Report n° 533, June 1986.

[8] P. CASPI, N. HALBWACHS, D. PILAUD, J.A. PLAICE : *LUSTRE : A declarative language for programming synchronous systems* ; 14th ACM Symp. on Principles of Programming Languages, Munich, January 1987.

[9] W.W. WADGE, E.A. ASHCROFT : *Lucid, the Dataflow Programming Language* ; Academic Press U.K., 1985.

[10] A. BENVENISTE, P. LE GUERNIC : *A denotational theory of synchronous communicating systems* ; INRIA, Rennes, Research Report n° 685, June 1987.

[11] S.J. YOUNG : *Real time languages : design and development* ; Elis Horwood publ., 1982.

[12] G. KAHN : *The semantics of a simple language for parallel programming* ; Information Processing 74, J.L. Rosenfeld, editor, North-Holland, 1974, 471 – 475.

[13] D. HAREL, A. PNUELI : *On the Development of Reactive Systems* ; Logics and Models of Concurrent Systems, NATO ASI Series, Vol. F13, Springer-Verlag, 1985, 477 – 498.

[14] T. GAUTIER, P. LE GUERNIC, A. BENVENISTE, P. BOURNAI : *Programming real-time with events and data flow* ; Information Processing 86, Proceedings of the IFIP 10th World Computer Congress, September 1986, H.-J. Kugler, editor, North-Holland, 1986, 469 – 498.

[15] B. LE GOFF, P. LE GUERNIC : *Granules' help to sharing out a data flow graph over multiprocessor hardware* ; forthcoming paper (digest to appear in Proceedings of STACS 88, Springer-Verlag).

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