The signal software environment for real-time system specification, design, and implementation

Albert Benveniste, Paul Le Guernic, Christian Jacquemot

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 1105

Programme 5
Automatique, Productique,
Traitement du Signal et des Données

THE SIGNAL SOFTWARE ENVIRONMENT FOR REAL-TIME SYSTEM SPECIFICATION, DESIGN, AND IMPLEMENTATION

Albert BENVENISTE
Paul LE GUERNIC
Christian JACQUEMOT

Octobre 1989

RR-1105

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

The SIGNAL software environment for real-time system specification, design, and implementation.

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

Christian JACQUEMOT
CNET PAA/OGE/SML, 3 Av. de la République
92131 ISSY-LES-MOULINEAUX, FRANCE

Publication Interne n° 490 - Septembre 1989 - 34 Pages

ABSTRACT

In this paper, "reactive systems", i.e. systems which interact permanently with their environment, are considered. Such systems are encountered, for instance, in real-time control or signal processing systems, Command-Control-Communication systems, man-machine interfaces, to mention just a few. We present the SIGNAL software environment designed and developped at INRIA-IRISA and its formal calculus system to perform the above mentioned tasks. We outline the principles of the SIGNAL "synchronous" language and its calculus system, which is based on formal manipulations of nonlinear dynamical systems over the finite field of integers modulo 3. Then we present the prototype workstation designed at CNET, where automatic mapping of a SIGNAL program onto a multi-transputer target architecture is performed, with the aid of a sophisticated graphic environment. Finally, we discuss what could be the impact of such tools on the future of real-time systems specification, design, and implementation.

Le système SIGNAL pour le développement d'applications temps-réel

RESUME

Nous présentons une méthode de développement d'applications temps-réel s'appuyant sur le langage de programmation SIGNAL conçu et développé à l'INRIA-IRISA. Après un exposé des motivations qui soustendent cette méthodologie, nous présentons les principes du langage SIGNAL, puis nous décrivons le poste de travail SIGNAL développé au CNET.

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

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

The SIGNAL software environment for real-time system specification, design, and implementation.

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

Christian JACQUEMOT
CNET PAA/OGE/SML, 3 Av. de la République,
92131 ISSY-LES-MOULINEAUX, FRANCE

August 25, 1989

Abstract

In this paper, "reactive systems", i.e. systems which interact permanently with their environment, are considered. Such systems are encountered, for instance, in real-time control or signal processing systems, Command-Control-Communication (C^3) systems, man-machine interfaces, to mention just a few. The specification, design, and implementation of such reactive systems require the following tools :

. a concurrent programming or specification language;

. a powerful formal tool which is able to verify, prove the correctness of, or even synthetize the logic and synchronization mechanisms which should control the reactive system being designed;

. a powerful tool which is able to transform the hierarchical structure of the specification in order to match a particular target architecture.

In this paper, we shall present the SIGNAL software environment designed and developped at INRIA-IRISA and its formal calculus system to perform the above mentioned tasks. We shall outline the principles of the SIGNAL "synchronous" language and its calculus system, which is based on formal manipulations of nonlinear dynamical systems over the finite field of integers modulo 3. Then we shall present the prototype workstation designed at CNET, where automatic mapping of a SIGNAL program onto a multi-transputer target architecture is performed, with the aid of a sophisticated graphic environment. Finally, we shall discuss what could be the impact of such tools on the future of real-time systems specification, design, and implementation.

Contents

1 Introduction: reactive systems.

Developping complex real-time systems is generally considered as a major challenge for the overall area of information sciences (remember the debate about the feasibility of SDI). In this paper, we shall emphazise on how the "programming" of such systems should be performed, and why control science could have a significant impact on the future of this activity, although it is generally considered as relevant to computer science only.

1.1 Some application areas: the notion of "reactive" system.

Here are some typical application areas generally related to information processing.

- **Real-time large scale control systems.** Here, the main challenge is to provide a tool which is flexible enough to allow an easy specification, and powerful enough to guaranty that the actual implementation meets the specifications.

- **Data communication systems** are prototype of real-time signal processing systems. The main difficulty is to provide aids for a highly efficient implementation (in terms of throughput), so that algorithms *and* machine architectures should be considered within the same framework.

- **Radar, sonar, and more generally C^3-systems.** Here, all the above mentioned difficulties are encountered. A first challenge is the systems specification and modification, a second one is to guaranty that the actual implementation meets the specification, and the third one is the importance and the complexity of the target architecture (generally a distributed system).

Real-time systems studies are usually composed of two kinds of investigations: 1/ how events, communications, and computations should be combined and ordered to perform the desired function, and, 2/ how fast can run a given implemented system. Here we do not consider the second aspect, and we rather concentrate on the first one. Hence our main concern is that the above discussed systems permanently *interact* between themselves and with their environment: for this reason, we shall merely refer in the sequel to *reactive systems*. The notion of reactive system was introduced in [11], and extensively used in [6].

1.2 Requests for reactive systems specification and design tools.

These requests were revealed by the discussion above, and are now listed:

1. The tool should provide all the following requested features to allow an easy specification:

   - since real-time is of major concern, timing, events, and synchronisation should be easily handled,
   - parallelism and hierarchical style should be promoted,
   - the same tool should encompass both synchronisation/logic , as well as computation flow,
   - the same tool should support both *algorithm* and *hardware* description and handling,
   - the tool should allow a *portable* design, which means that most of the task should be performed independently of the target architecture.

2. The tool should provide an extended capability of program manipulation and transform:

   - the mapping from the high level specification to the actual implementation on the target architecture should be as automatic as possible in order to avoid a distorsion to be introduced,
   - the tool should support *proof systems* or equivalent means to guaranty that the actual implementation meets the specification,
   - the user friend design method should not be provided at the price of an excessive overhead at the run time.

3. All that should be provided while taking into account the *asynchronous* nature of the communications with the external world, or between subsystems of a distributed system.

1.3 A quick look at the state of the art.

Direct programming of finite state machines is often used to program relatively small real-time kernels. Automata obviously yield excellent and measurable run-time efficiency, and non-trivial correctness proofs can be performed by automatic temporal-logic formula checkers such as EMC

[9], XESAR [14], or by automata observation systems such as AUTO [2]. However, the human design and maintenance of automata turns out to be very difficult and error-prone: request 1. is certainly not satisfied.

Sequential tasks runing under a "real-time" operating system are widely used. This is often the approach followed for instance when several DSP processors are used. Today, DSP processors are highly powerful number crunching machines that are rather easy to use as singletons, thanks to reasonably high level programming environment. However, apart from the *Transputer*, by INMOS, where OCCAM [13] is available as a concurrent programming language, no powerful facility is provided for multi-processor programming. However, in any case, request 2. above is not satisfied.

Petri-Net based tools [10] and **graphical languages** such as "signal flow graphs" are often used in process control or signal processing systems specification. The former ones are devoted to the specification of sequential tasks and synchronisation. The latter ones are suitable to the specification of low level regulators or signal processing systems; unfortunately, they do not allow for any kind of flexibility related to event driven control ("`while`" statements for instance). The joint use of the two approaches is rather difficult, both from the user viewpoint (request 1.) and from the implementation viewpoint (request 2.).

Concurent high level programming languages, such as ADA [1] allow hierarchical and modular program development. Their tasking mechanism and communication primitives are defined at the language level, and are portable. However, the actual implementation of a given program depends on the particular target machine, so that no proof system is available concerning the synchronisation aspects. On the other hand, any parallelism in the specification will result in an overhead at the run time, even if the target machine is of Von Neumann type: request 2. is still not satisfied.

As a conclusion, most existing methods take into account the request 3. above, at the loss of (at least) request 2. Exceptions are the direct specification of automata, or the signal flow graphs, but none provide the desirable

flexibility as stated in the request 1.

2 Reactive systems and the principle of synchronicity.

2.1 An immediate overall objective.

This objective is illustrated on the diagram of the figure 1. Notice that all the requests 1. to 3. are apparently needed for this objective to be satisfied.

2.2 The principle of synchronicity: synchronous languages.

The family of the *synchronous languages* (the imperative language ESTEREL [6], the data-flow functional language LUSTRE [8], and the block-diagram oriented relational language SIGNAL [3] [5]) have been introduced based on the following fundamental remark: requests 1. to 3. are *not compatible,* it is not possible to fullfil all of them simultaneously. On the other hand, it is judicious *to ignore request 3.,* since *requests 1. and 2. will in turn be satisfied!*

Let us explain how request 3. can be discarded on a simple (but still relevant) example illustrated on the figure 2. Consider a reactive system with two external inputs:

- a sampled signal y delivered by some sensor
- an interrupt signal s indicating some exception to be handled.

Strictly speaking, these two inputs are received *asynchronously,* which means that deciding upon their mutual interleaving is (slightly) arbitrary. Assume a (very small) real-time kernel is available, which decides this for us. This real-time kernel delivers flashes or *events* composed of pairs of one of the following forms:

- $(y, nothing)$ (*nothing* is denoted by a diamond on this figure): the signal has been received without any interrupt

- $(nothing, s)$: an interrupt occurred between two successive receptions of the signal
- (y, s): an interrupt occured "almost simultaneously" with the signal's reception, so that the real-time kernel considered them as simultaneous.

By the way, this real-time kernel transformed the *asynchronous* perception of time (i.e. local to each input port, figurated by the grey halo locally around each channel on the figure 2) into a *synchronous* one (i.e. global to the two ports, figurated by the global grey halo on the figure 2). Such a transformation is up to a certain extend arbitrary. However, if the *events* provided by the real-time kernel are now considered as the *inputs* of our reactive system, then *significant transforms on the system can be performed with the guaranty of equivalence preserving. And this is the basis for the fullfilment of the requests 1. and 2. above.* A good way to get the intuition of why such transformations will now be possible is to convince yourself that, thanks to the synchronicity assumption, a reactive system is nothing but a *system of dynamical equations* where some signals may have the distinguished status *absent.* And it is likely that a little mathematics will provide us with this ability to transform systems while guarantying equivalence. We shall now concentrate in the sequel on the SIGNAL language.

3 The SIGNAL language.

3.1 An informal introduction.

For a complete presentation and motivation of the language, we refer the reader to [3], [4], and [5]. SIGNAL handles (possibly infinite) sequences of data with time implicit: such sequences will be referred to as *signals.* For example, $\mathbf{x}$ denotes the infinite sequence $\{x_t\}_{t\geq 1}$ where the time index t is attached to this signal; generally, different signals possess different time indices (remember the example of the figure 2). We shall term *clocks* the different time indices that occur in a reactive system: clocks are attached to signals. *A* SIGNAL *program specifies relations, i.e. dynamical constraints,*

on a set of signals. These constraints involve both the clocks, and the values of the signals when they are present.

3.1.1 A small example.

Here follows a first example:

```
ASYNCH_SWITCH {!  bool STATE }
      =
|    STATE := (not STATE) $ init false
end
```

This is a very simple SWITCH which is equivalent to the dynamical system:

$$\begin{aligned} STATE_t &= \text{not } STATE_{t-1} \\ STATE_0 &= \texttt{false} \end{aligned}$$

As we indicated before, time is implicit in SIGNAL. In this program, the symbol $ is a keyword of the language: this is just the z^{-1} shift operator, which implicitly acts according to the clock of the considered signal. The program is composed of an *interface* ASYNCH_SWITCH {! bool STATE } and its *body* { = ... end } . The interface specifies the *name* of the program and the list of *inputs* { ? list } and of outputs { ! list } . As one can see, the program above possesses no input: it only specifies a constraint on the signal STATE. Hence, SIGNAL is a *relational* language. Here, nothing is specified about the precise instant at which the switching occurs, hence the prefix ASYNCH_ in its name.

The following program specifies that switching must occur when the pure event (a boolean which is always true) TOP is received, so that the result is now an input-output map:

```
SWITCH {?  event TOP ! bool STATE }
      =
|    STATE := (not STATE) $ init false
|    synchro STATE, TOP
end
```

The communication operator "|" specifies that the signals STATE in both instructions are identical. The instruction synchro ...,... forces the listed signals to have the same clock. In this program, access to the STATE is possible only when a switch occurs (i.e. at any of the instants of the clock of this signal). If it is desired to read more often the internal state of the switch, the following program can be built:

```
READ_SWITCH {?  event READ, TOP ! bool ON_OFF }
      =
|    SWITCH
|    ON_OFF := STATE default (ON_OFF $ init false)
|    synchro ON_OFF, (TOP default READ)
end
```

In this form, the program is considered as a block-diagram, where the linking operator "|" connects the primitive instructions or the sub-blocks. Its meaning is the following: ON_OFF is read, either when TOP is received (a switch occurs) or when READ is received. In the former case, ON_OFF takes with priority the new value of STATE, while in the latter case ON_OFF takes by "default" its previous own value, so that the last value of STATE is emitted in this case. At this stage *we have at hand a switch with a state that can be read whenever needed.*

The next step in our example is a counter with reset:

```
COUNT {?  event SEC, RESET ! integer N }
   =
|    N := (0 when RESET) default ((N $ init 0) + 1)
|    synchro N, SEC
end
```

The instruction N := ... resets the counter to the value 0 when the pure event RESET is received. By "default", the counter is incremented with the constant quantity 1. This instruction fires the counter at least when RESET is received, but the default instruction allows arbitrary additional firings.

The instruction synchro specifies that N is incremented and delivered exactly when RESET is received or when SEC is received: the counter counts the occurrences of SEC (let's say this corresponds to the seconds).

Finally, here follows a simple *timer* which uses the programs above. In this program changes of names in the i/o signals of the sub-blocks is performed in order to achieve the interconnection of these sub-blocks. For instance, ? READ:SEC means that the input READ of READ_SWITCH is renamed SEC: the result is that COUNT and READ_SWITCH POSSESS SEC as common input.

```
TIMER {?  event SEC, RESET , START_STOP ! integer TIME}
      =
|    COUNT
|    READ_SWITCH ? TOP:START_STOP, READ:SEC
|    TIME := N when ON_OFF
end
```

The button START_STOP switches between the on and off states, the time is delivered only in "on" state, i.e. when ON_OFF is true. The reception of SEC both causes the increment of the counter and the reading of the current state ("on" or "off") of the timer. The RESET signal causes the reset of the timer. Referring to the figure 2, this example possesses one regularly sampled signal, and two interruption signals.

3.1.2 Programming this example on the IRISA SIGNAL system.

We have used the block-diagram oriented interface developped for SIGNAL at IRISA to program this example. The figures 3–4 show a part of the graphical programming of the program READ_SWITCH. The figure 5 shows the corresponding full-text source code, produced by the system. The syntax shown here is more loquacious than the one we did use: this is the syntax of an older version, but the two programs are equivalent. In particular, the type "event" is not available in this version, so that signals of type event are here declared as "logical", and the instruction event B produces the clock of the logical signal B.

Then the figure 6 shows an intermediate level code which is produced by the compiler according to the principles we shall outline next. This intermediate level code is ready for sequential or parallel final code generation, and is given in the form of another SIGNAL program, which can be returned to the programmer for debugging.

3.2 Discussion.

Although rather complex in synchronisation as compared to computation, this example reveals the features of SIGNAL.

- SIGNAL is a block-diagram oriented language (request 1.) which is able to handle both numerics and synchronisation (request 1.). A quick inspection reveals that block-diagrams generally specify relations rather than i-o maps: hence SIGNAL is a *relational* language.

- Clocks are assigned to signals. SIGNAL programs can receive as inputs signals with different clocks. The synchronisation mechanisms are specified via *local* relations involving a few (2 or 3) signals, and the global synchronisation that results is *synthetized* by the compiler as we shall see next (request 1.).

- The SIGNAL programming style is definitely oriented to the specification of synchronisation constraints on signals and their clocks. By the way, key synchronisation properties that should be satisfied by the reactive system can just be included as particular instructions in the program: they will be automatically satisfied at the run time. Hence, SIGNAL allows a **self-proved style of specification and programming** (request 2.).

- Thanks to the above features, SIGNAL can also be used as a hardware description language to specify the functional behaviour of architectures.

To justify the above claims concerning the self-proved programming style, as well as to show why the request 2. is also fullfiled, we need to describe briefly the SIGNAL formal calculus system, which is the basis the the SIGNAL compiler.

4 An outline of the SIGNAL formal calculus system.

It is shown in [3] that the following five instructions are sufficient to specify any reactive system:

```
p(b1,...,bn), y := f(x1,...,xn)
y := x $ init y0
y := x when b
y := u default v
P|Q
```

We shall now comment these instructions, and introduce at the same time their *encoding* using dynamical systems over the finite field $\mathcal{F}_3$ of integers modulo 3. This coding will be the basis of SIGNAL's formal calculus system which is the core of the compiler.

Instructions `a and b = true , y := u+v`
The first one is a particular case of the generic instruction `p(b1,...,bn)` where `p(...)` is any relation on booleans. It extends to signals the mentioned relation. First, the signals `a, b` must *have the same clock* (i.e. they must be based on the same time index), and second, when they are present, their actual values must satisfy the mentioned relation on boolean values.

The second instruction is a particular case of the generic form `y := f(x1,..,xn)`. It extends to signals the mentioned function on data. First, all signals `y, u, v` must have the same clock, and second, when they are present, the value carried by `y` must be equal to `u+v`.

The reasoning mechanisms of SIGNAL can handle (i) the presence/absence, (ii) the boolean values since they are important in modifying clocks, and (iii) *dependence graphs* to encode data dependencies in non boolean functions in order to avoid to solve general implicit systems of equations. Hence three labels are needed to encode $absent, true, false$: the finite field $\mathcal{F}_3$ is

used for this purpose:

$$absent \rightarrow 0,\ true \rightarrow +1,\ false \rightarrow -1$$

Using this mapping, the two instructions `a and b = true , y := u+v` are respectively encoded as follows:

$$a^2 = b^2,\ ab + a + b = 0 \quad (1)$$

$$y^2 = u^2 = v^2,\ \mathtt{u} \xrightarrow{y^2} \mathtt{y} \xleftarrow{y^2} \mathtt{v} \quad (2)$$

In these equations, the variables $a, x, \ldots$ refer to infinite sequences of data in $\mathcal{F}_3$ with time index implicit. The first equation of (1) expresses that the two signals `a` and `b` must have the same clock, while the second one encodes the particular boolean relation (the first equation is here a consequence of the second one). The first equation of (2) again expresses that all signals must have the same clock, while the labelled graph expresses that the mentioned data dependencies hold when $y^2 = 1$, i.e. when all signals are present (this is referred to as the *conditional dependence graph* since signals may be related via different dependencies at different clocks).

Since our purpose here is not to fully investigate the interconnection of clocks and dependencies (see [3], [4], and [5] for an extensive discussion on this topic), we shall concentrate in the sequel on *boolean* instructions.

Instruction `y := x $ init y0`

This instruction simply means $\forall t : y_t = x_{t-1}, y_0 = \mathtt{y0}$ where the time index t enumerates the instants where both signals `x, y` are present: this is just a shift register. Boolean shift registers are encoded as follows

$$\xi' = (1 - x^2)\xi,\ \xi_0 = \mathtt{y0} \quad (3)$$

$$y = x^2\xi \quad (4)$$

where the variable ξ is the state, and ξ' denotes its next value according to any (hidden) time index which is more frequent than the index t above. This is a nonlinear dynamical system. Notice that receiving $x = 0$ (i.e. the input being absent at the considered instant) does not change the state nor produces any output (since $y = 0$ holds also in this case).

Instruction `y := x when b`.
The value of `x` is delivered at the output `y` when `x` and `b` are present, and `b`$= true$. This gives:

$$y = x(-b - b^2) \tag{5}$$

Instruction `y := u default v`.
The value of `u` is delivered at the output `y` when `u` is present, otherwise the value of `v` is delivered if `v` is present, otherwise nothing is delivered. This gives the coding:

$$y = u + v(1 - u^2) \tag{6}$$

Instruction `P|Q`
Here, `P`, `Q` are names of SIGNAL *programs*, i.e. already defined sets of instructions or programs, and `P|Q` denotes the new program obtained by combining the already defined ones by considering that signals with identical names are identical (just as in mathematics). The coding of this instruction is obtained by considering together as a single dynamical system the two dynamical systems associated to `P` and `Q`.

The general form. By combining together the particular dynamical systems of eqns (1) (2) (4) (5) (6), we obtain the general mixed form of dynamical system over $\mathcal{F}_3$ and labelled graphs (this is referred to as *dynamical graphs* in [5]):

$$\begin{cases} X_{t+1} &= P(X_t, Y_t) \\ 0 &= Q(X_t, Y_t) \end{cases}$$

$$\mathtt{Y(i)} \overset{H(i,j)}{\rightarrow} \mathtt{Y(j)}$$

where X, Y are vectors with components in $\mathcal{F}_3$, P and Q denote polynomial vectors on the components $X(i), Y(j)$ of X, Y. The components of X are the states of the boolean registers, and the components $Y(j)$ of Y are the encoding in $\mathcal{F}_3$ of all the signals `Y(j)` involved in the program. The time index t is any time index which is more frequent than the clocks of all the components of `Y`. The last equation specifies a labelled directed graph, which encodes the data dependencies (here, from the i-th component of `Y` to the j-th one) labelled with the clocks at which these dependencies hold

(here, $H(i,j)$ is a polynomial expression in $\mathcal{F}_3$, and $H(i,j) = 1$ specifies the considered clock).

It is shown in [4], [3], [5], and extensively in [12] how this coding can be used, with the help of polynomial ideal theoretic methods, to answer for instance the following kind of questions:

- Is a given SIGNAL program a relation, or an input-output map? Since only i-o maps can be ultimately executed, this is a fundamental question that has to be solved prior to compilation.
- Do some instructions contradict other ones? This is the basis for proving program correctness, since key properties that are desired to be satisfied can be simply included as additional constaints written in SIGNAL (here we use the relational nature of the language).
- Transform a given SIGNAL program written in a relational form (say, via a block-diagram) into an *executable* form, namely a "master" finite state machine which fires "slave" computations. The result of this transform may either have the form of a *sequential* machine, or still possess a certain amount of *parallelism*. This latter feature is a key one for the fullfilment of request 2.

5 CNET's SIGNAL workstation.

The overall purpose of this programming environment is to provide an integrated chain with SIGNAL as task specification language, and a *Transputer* based multiprocessor target architecture.

The development of a real-time application with this environment is performed as follows:

1. Specify the application using the *graphical* SIGNAL editor. This editor allows to draw programs using a graphical interface based on the "Mac-Intosh" philosophy. Services are provided to facilitate the browsing in the hierarchy of the block-diagram which specifies the SIGNAL program. The POSTSCRIPT copy of a screen corresponding to such a program edition is shown in the figure 7.

2. Use the SIGNAL compiler to generate standard C code, based on the principles of compilation we have outlined before. At this stage, the SIGNAL formal calculus system is used to verify the correctness of the program synchronization and logic with respect to the specification.

3. Execute this C code on the host machine (SUN 3/.. with SUN-VIEW, or a UNIX machine with X-VINDOW V11), and use again the graphical interface to see the results of a (lower speed) execution. This is done by selecting desired ports with the mouse, and opening various styles of views showing the results edited in real time. This is shown in the figure 8. This allows to debug the numerical aspects of the application, and to verify that the desired processing is conveniently performed.

4. Use the graphical interface to describe the configurable target architecture. Building blocks of this architecture are Transputers (T800, T414, T212,...), crossbars C004, communication channels, and the host. In the actual architecture, the Transputers in charge of the computations are powered by TMS320C** by Texas Instruments. An example is shown in the figure 9.

5. Use the same graphical editor for the SIGNAL program and the described target architecture, and allocate the SIGNAL modules to the various Transputers by placing program boxes into architecture boxes (using the mouse as in a Mac-Intosh). Then, use again the SIGNAL compiler to generate OCCAM code: the parallel structure of this code matches the parallel structure of the target architecture (no internal communication occurs inside a given Transputer during the execution, so that the overhead due to parallelism is minimised, cf request 2.). The same graphical facilities as before can be used for the real-time execution of the resulting distributed program.

This software corresponds to 70 000 lines of C code. The C source code has been generated automatically from a SMALLTALK-80 code (35 000 lines). This method combines the advantages of prototyping in SMALLTALK-80 and executing the software in C.

6 Conclusion: the future of real-time systems specification, design, and implementation.

6.1 A new look at this topic.

Here follow the items that are involved in the development of reactive systems (cf. figure 10):

- the SIGNAL program which specifies the task to be realized,
- the actual target architecture,
- how act the communication links which exist between the distributed components of this target architecture (channels, buses, ...),
- how the whole system does interact with the external world (sensors, interruptions, actuators,...),
- what are the protocols that are used to compensate for the "desynchronising" effects of all these communications.

The asynchronous nature of internal and external communications mainly results in the fact that *when* data enter the considered channels *does not* completely determine *when* they exit. In the framework of computer science, this is known as a *non-deterministic* behaviour. In the control science framework, we consider that the considered channel *does not act as an input/output map* as far as the synchronisation aspects are concerned. Specific channels are only characterized by specific *constraints*: for instance, the order of the data is not reversed, at most 1 data may be present in the channel at a given instant, etc... But such channel characteristics are perfectly modelled as SIGNAL programs that are *not* i/o maps, such programs will be called "relational". Hence, SIGNAL can be used for all items above as follows:

- *the* SIGNAL *program which specifies* the task to be realized,

- the actual target architecture, *reflected in the hierarchy of the* SIGNAL *program above,*
- how act the communication links which exist between the distributed components of this target architecture (channels, buses, ...), *modelled by "relational"* SIGNAL *programs,*
- how the whole system does interact with the external world (sensors, interruptions, actuators,...), *modelled by "relational"* SIGNAL *programs,*
- what are the protocols that are used to compensate for the "desynchronising" effects of all these communications, also *modelled by* SIGNAL *programs.*

The resulting program will be termed the *Implementation Modelling Program.* Notice that this is exactly the usual approach followed in control science when the synthesis of a control law is desired: *using a simulator of the plant and the designed control law, investigate how they interact together.* Using this approach, several questions could be answered thanks to the SIGNAL formal calculus system:

- Can we prove that the actual implementation meets the specification? More precisely, can we prove that, if all signals involved in the modelling of the channels and protocols are hidden in the Implementation Modelling Program, the result is equivalent to the original specification? A similar task is for instance performed in [7] based on an automata theory oriented mathematical model.
- Assume that the communication protocols have not yet been defined, can we *synthetize* protocols guarantying the above equivalence? This is very close to the "exact model following control" problems that are analysed in [12].

6.2 Conclusion

As a conclusion, we would like to stress the following points.

- Real-time "programming" should be considered as a vanishing activity in a short range future: *specification and verification* is now at hand, thanks to the approach of the *synchronous* languages (ESTEREL, LUSTRE, and SIGNAL) introduced above.

- Long range effort should move towards a *synthesis* of such systems. And control science *is concerned* with this research area, as the example of the SIGNAL language revealed. A possible point of view on this topic has been presented in this paper via the study of "control problems" for implicit nonlinear dynamical systems over finite fields. This issue is investigated in details in [12].

References

[1] ADA, *The programming language* ADA *reference manual,* Springer Verlag, LNCS 155, 1983.

[2] D. VERGAMINI, "Verification by means of Observational Equivalence on Automata", INRIA report 501, 1986.

[3] A. BENVENISTE, B. LE GOFF, P. LE GUERNIC, "Hybrid Dynamical Systems theory and the language SIGNAL", INRIA report 838, 1988.

[4] A. BENVENISTE, P. LE GUERNIC, "Hybrid Dynamical Systems theory and nonlinear dynamical systems over finite fields", CDC 88, 209-213, Austin, 7-9 Dec. 1988.

[5] A. BENVENISTE, P. LE GUERNIC, C. JACQUEMOT, "Synchronous programming with events and relations: the SIGNAL language and its semantics", IRISA report 459, 1989.

[6] G. BERRY, G. GONTHIER, "The ESTEREL synchronous programming language: design, semantics, Implementation", INRIA report 842, 1988, to appear in *Science of Computer Programming.*

[7] G. BOUDOL, R. DE SIMONE, D. VERGAMINI, "Experiment with AUTO and AUTOGRAPH on a simple case of sliding window protocol", INRIA report 870, 1988.

[8] P. CASPI, D. PILAUD, N. HALBWACHS, J.A. PLAICE, "LUSTRE, a declarative language for programming synchronous systems", Proc of the 14th ACM symp. on POPL, 1987.

[9] E.M. CLARKE, E.A. EMERSON, A.P. STISLA, "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Trans. on Programming Languages and Systems, 8, 2,* (1986), 244-263.

[10] M. BLANCHARD, *Comprendre, maitriser et appliquer le* GRAFCET, Cepadues Editions, 1979.

[11] D. HAREL, "STATECHARTS: a visual approach to complex systems", *Science of Computer Programming,* 8 (3), 231-275, 1987.

[12] M. LE BORGNE, A. BENVENISTE, P. LE GUERNIC, "Polynomial Ideal Theory methods in DEDS and HDS", Proc. of the CDC 89, Tampa, 1989.

[13] INMOS LTD, *The* OCCAM *programming manual,* Prentice Hall, 1984.

[14] J-P QUEILLE, J. SIFAKIS, "Specification and verification of concurrent systems in CESAR", Proc. Int. Symp. on Programming, Springer Verlag, LNCS 137, 1982.

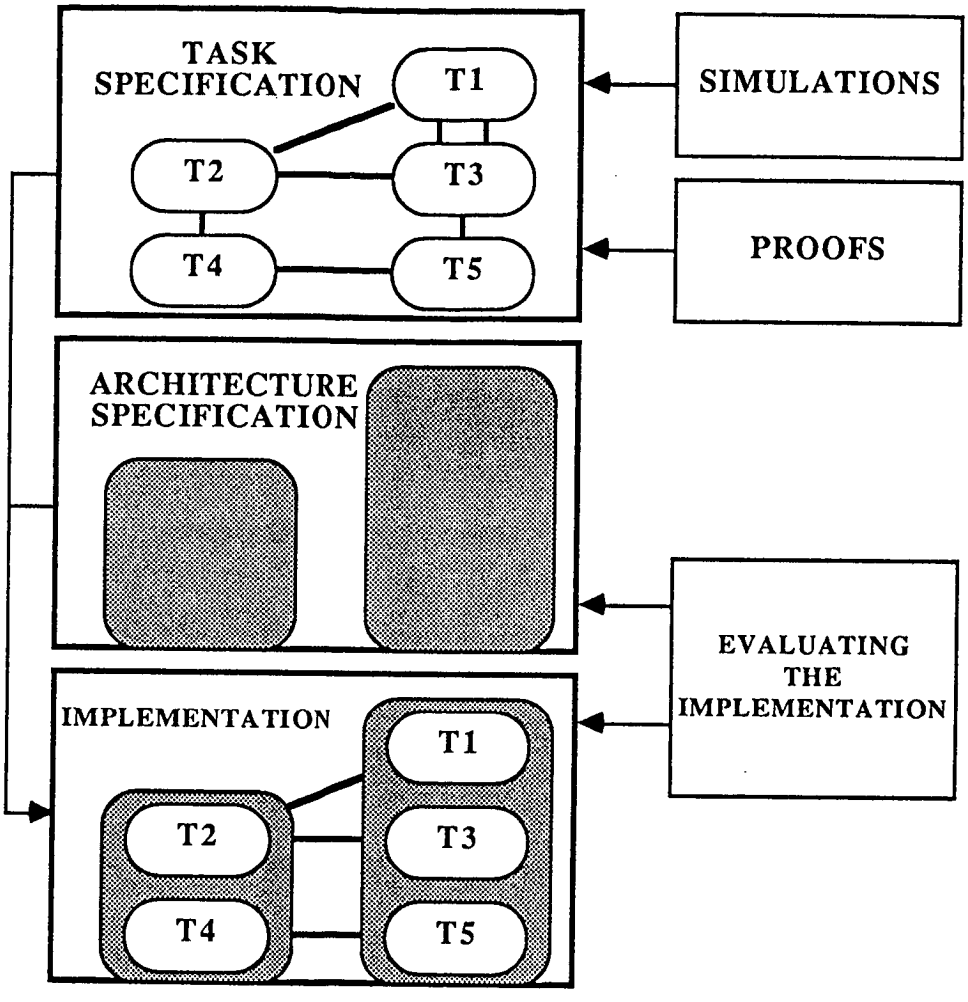

Figure 1: Reactive system design methodology.

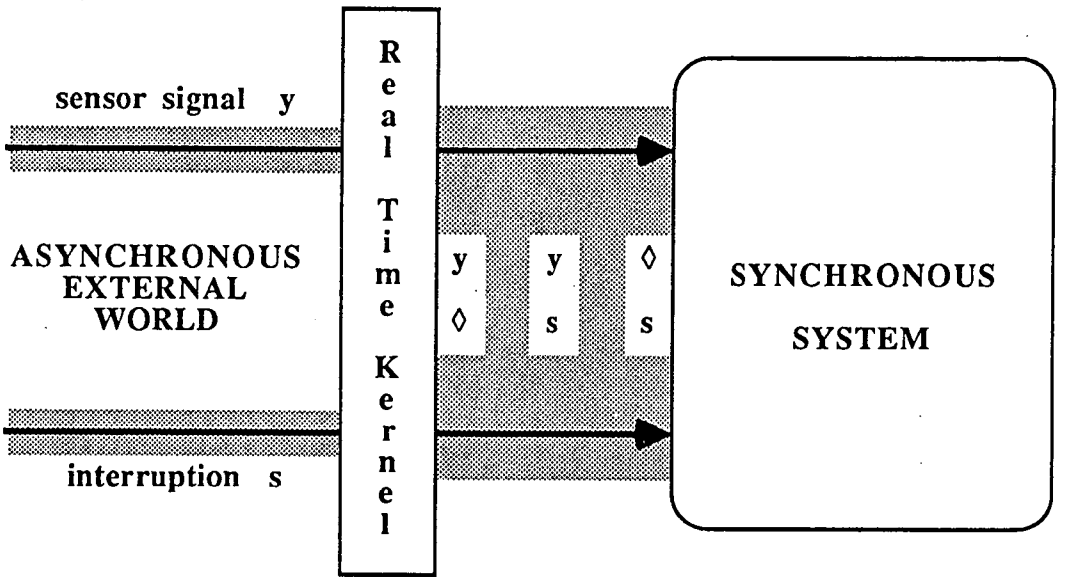

Figure 2: Asynchronous/synchronous transducer.

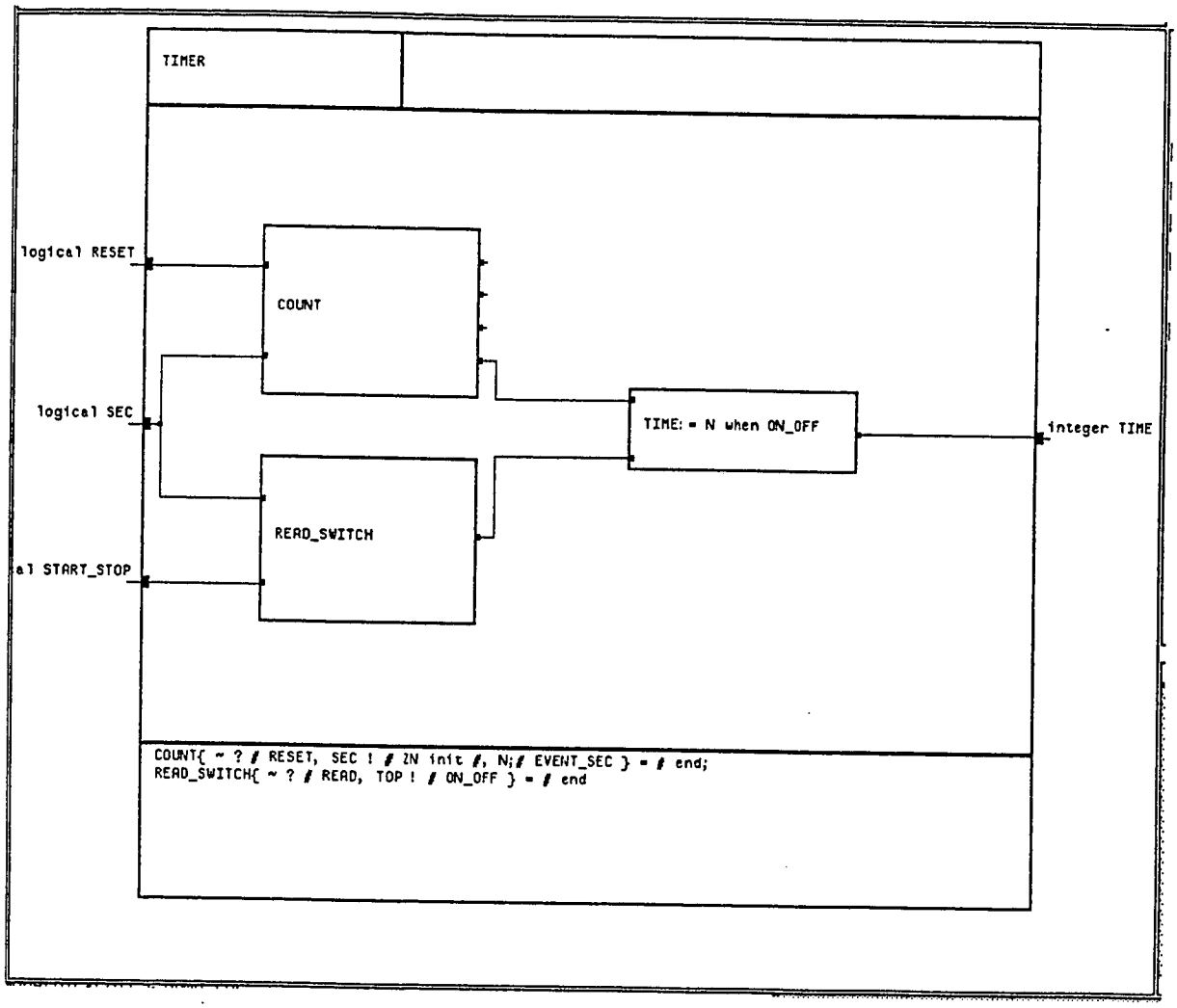

Figure 3: Programming the TIMER: highest level block-diagram.

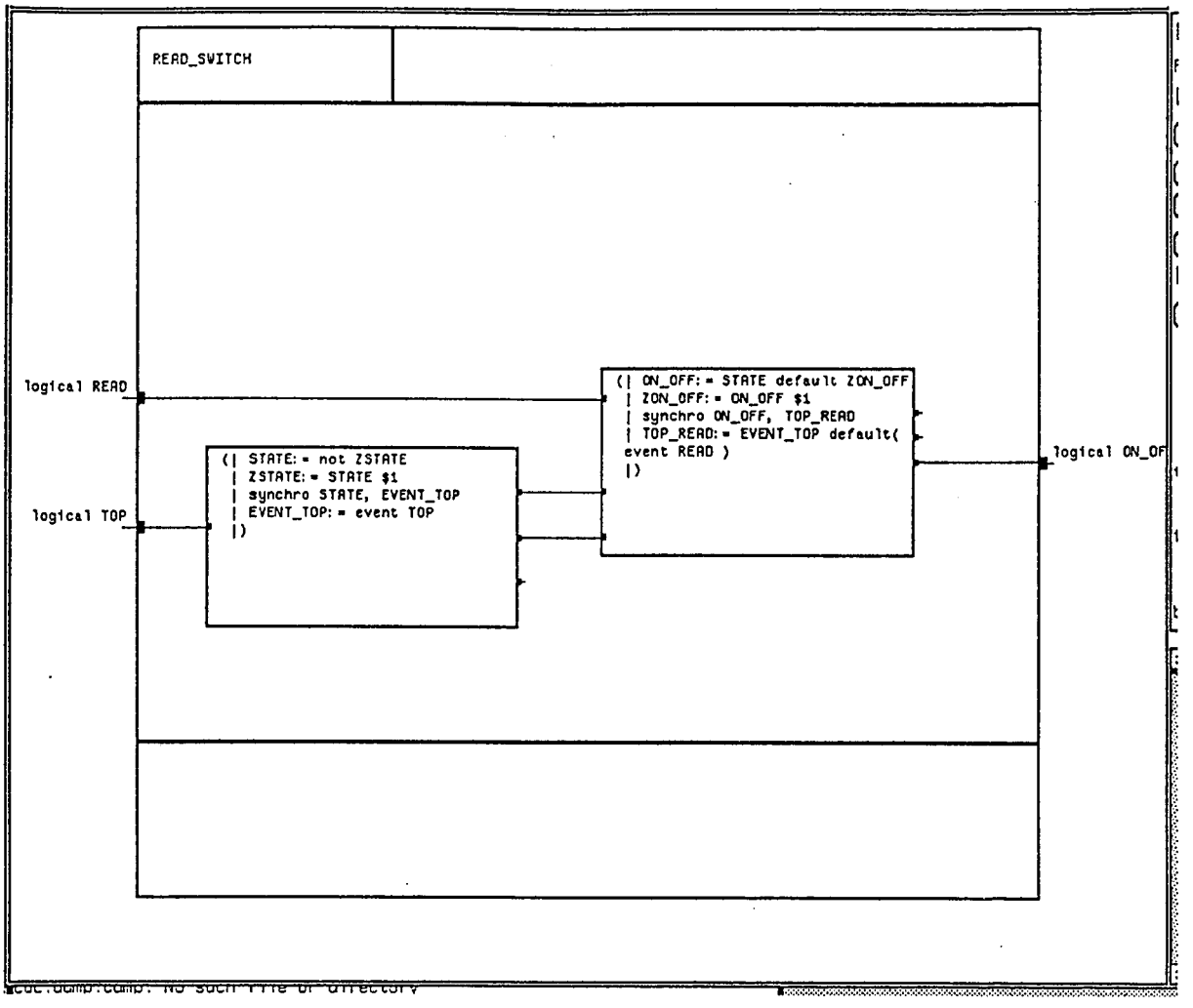

Figure 4: Programming the TIMER: block-diagram of the module READ_SWITCH.

```
TIMER{ ? logical RESET, SEC, START_STOP
       ! integer TIME }
=    (| COUNT ! N:N_1
            / EVENT_SEC, EVENT_RESET, ZN
     | READ_SWITCH ? TOP:START_STOP, READ:SEC
                   ! ON_OFF:ON_OFF_2
     | (TIME:= N when ON_OFF)? ON_OFF:ON_OFF_2, N:N_1
     |)/ ON_OFF_2, N_1
where
     logical ON_OFF_2, EVENT_SEC, EVENT_RESET;
     integer N_1, ZN init 0
process
     COUNT{ ? logical RESET, SEC
            ! integer ZN init 0, N;
              logical EVENT_SEC }
     =    (| N:= ( 0 when( event RESET ))default( ZN+1 )
          | ZN:= N $1
          | synchro N, EVENT_SEC
          | EVENT_SEC:= event SEC
          |)
     end;
     READ_SWITCH{ ? logical READ, TOP
                  ! logical ON_OFF }
     =    (| (| ON_OFF:= STATE default ZON_OFF
               | ZON_OFF:= ON_OFF $1
               | synchro ON_OFF, TOP_READ
               | TOP_READ:= EVENT_TOP default( event READ )
               |)? STATE:STATE_2, EVENT_TOP:EVENT_TOP_1
                 / ZON_OFF, TOP_READ
          | (| STATE:= not ZSTATE
               | ZSTATE:= STATE $1
               | synchro STATE, EVENT_TOP
               | EVENT_TOP:= event TOP
               |)! STATE:STATE_2, EVENT_TOP:EVENT_TOP_1
                 / ZSTATE
          |)/ STATE_2, EVENT_TOP_1
     where
          logical ZSTATE init false, ZON_OFF init false, TOP_READ, STATE,
                  STATE_2, EVENT_TOP, EVENT_TOP_1
     end
end
```

Figure 5: The full source program TIMER.

```
TIMER_TRA{ ? logical RESET_1, SEC_2, START_STOP_3
           ! integer TIME_4 }
=     (| (| synchro H_5_H, RESET_1
         |)
       | (| synchro H_24_H, START_STOP_3, STATE2_21
         | H_24_H
         |)
       | (| H_10_H:=H_5_H default H_10_H
         | synchro H_10_H, SEC_2, N1_11
         | H_10_H
         | (| H_8_H:=H_10_H when((not H_5_H)default H_10_H)
          |)
         | (| H_27_H:=H_10_H when H_25_H
          | synchro H_27_H, TIME_4
          | TIME_4:=N1_11 when H_27_H
          |)
         |)
       | (| H_17_H:=H_17_H default H_24_H
         | (| H_28_H:=H_10_H default H_24_H
          | synchro H_17_H, H_28_H
          |)
         | H_17_H
         | (| H_15_H:=H_17_H when((not H_24_H)default H_17_H)
          |)
         |)
       |) / H_27_H, H_25_H, H_17_H, H_15_H, H_10_H, H_8_H, N1_11, STATE2_21
where
     logical H_27_H, H_25_H, H_17_H, H_15_H, H_10_H, H_8_H;
     integer N1_11;
     logical STATE2_21
process
     H_24_H { ? logical H_24_H
              ! logical STATE2_21 }
     =     (| (| synchro H_24_H, ZSTATE_25
              | (| ZSTATE_25:=STATE2_21 $1
               | STATE2_21:=(not ZSTATE_25)when H_24_H
               |)
              |)
       |) / ZSTATE_25
     where
          logical ZSTATE_25
     end;
     H_17_H { ? logical H_24_H, H_17_H, H_15_H;
                logical STATE2_21
              ! logical H_25_H }
     =     (| (| synchro H_17_H, ON_OFF2_10, ZON_OFF_23
              | (| H_25_H:=true when ON_OFF2_10
               | (| ZON_OFF_23:=ON_OFF2_10 $1
                | ON_OFF2_10:=(STATE2_21 when H_24_H)default(ZON_OFF_23
                                when H_15_H)
                |)
               |)
              |)
       |) / ON_OFF2_10, ZON_OFF_23
     where
          logical ON_OFF2_10, ZON_OFF_23
     end;
     H_10_H { ? logical H_10_H, H_8_H, H_5_H
              ! integer N1_11 }
     =     (| (| synchro H_10_H, ZN_13
              | (| ZN_13:=N1_11 $1
               | N1_11:=(0 when H_5_H)default((ZN_13+1)when H_8_H)
               |)
              |)
       |) / ZN_13
     where
          integer ZN_13
     end
end
```

Figure 6: An intermediate level code for TIMER.

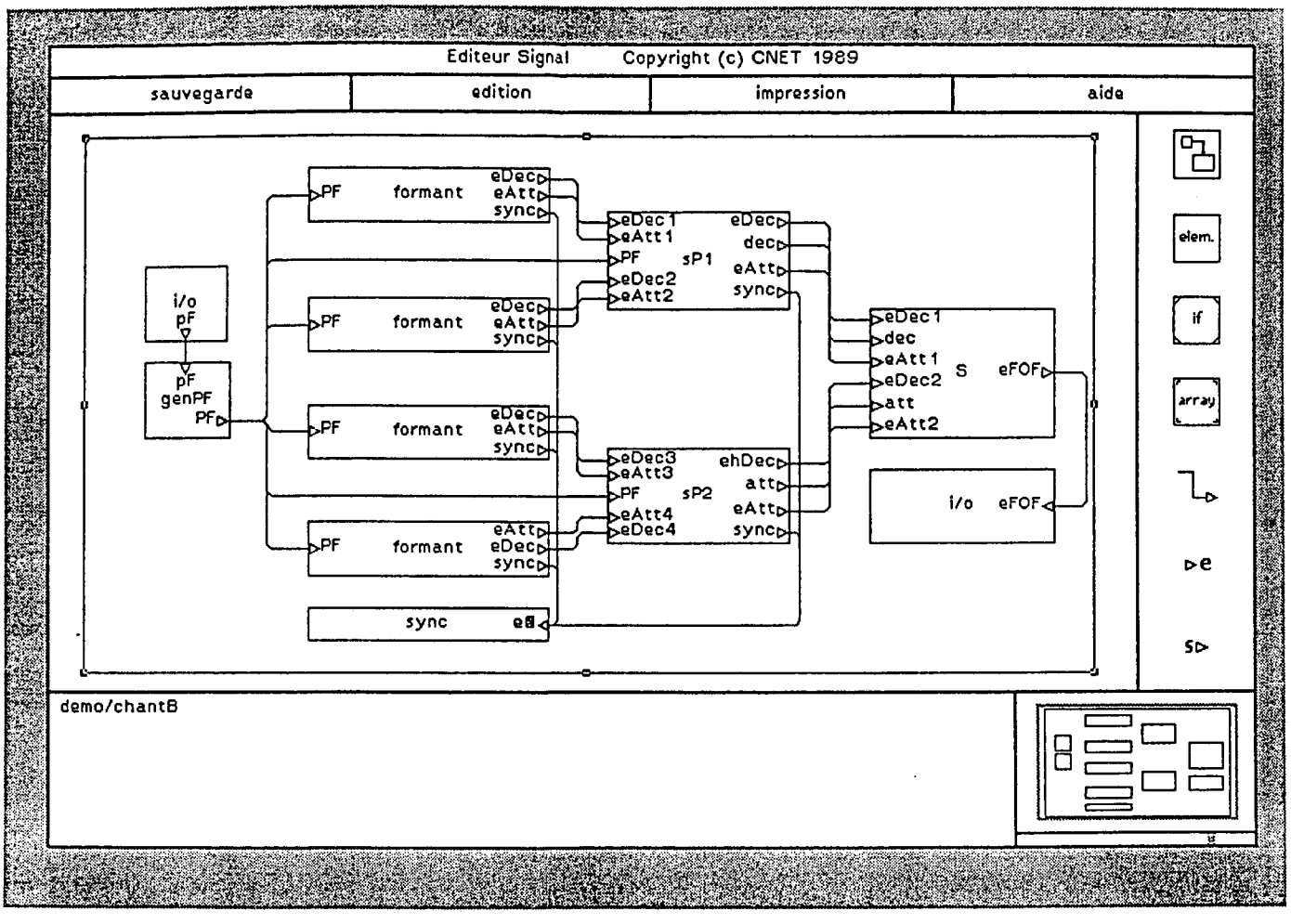

Figure 7: A view of the program "song"

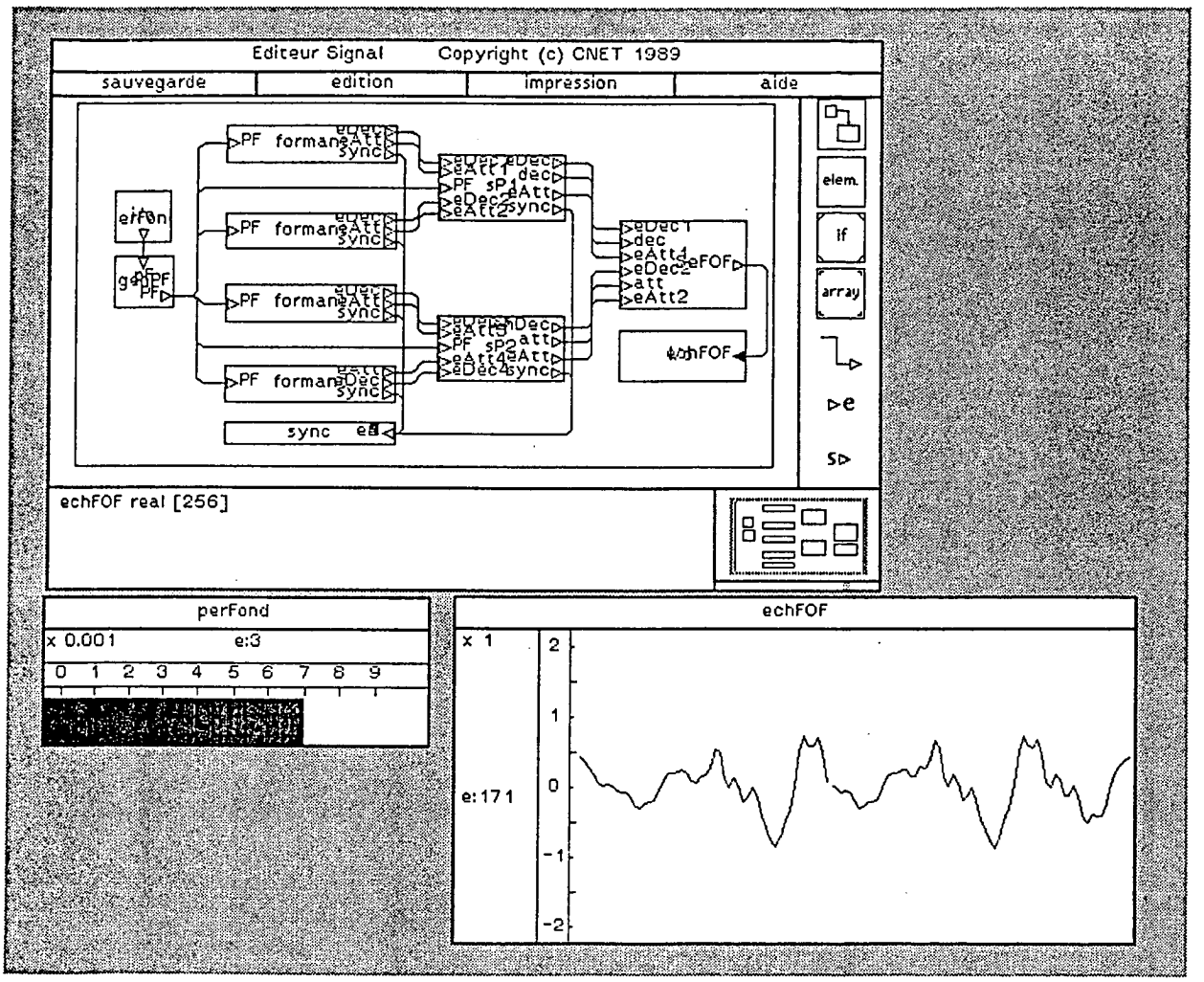

Figure 8: Running the program "song"

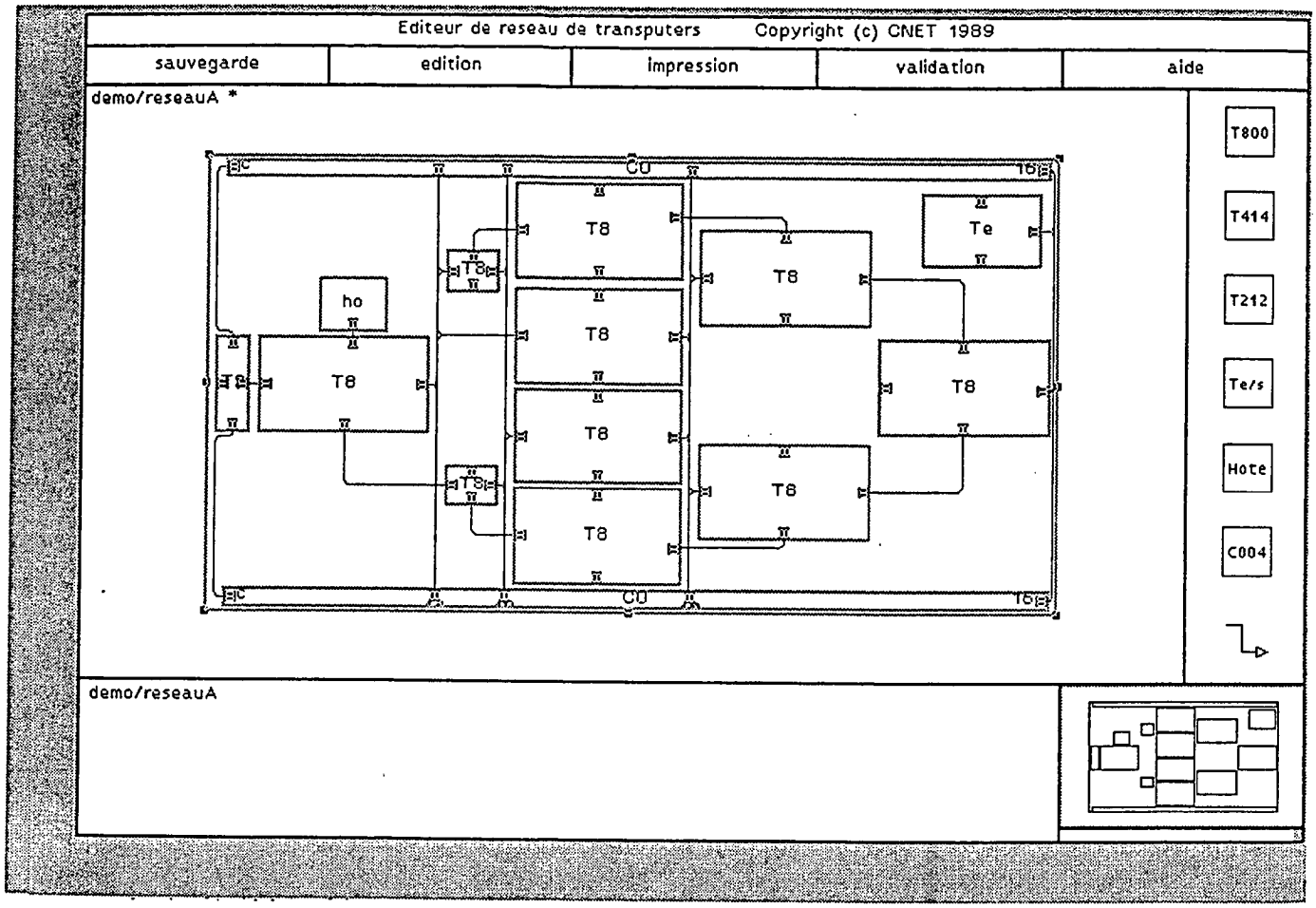

Figure 9: The specified multi-Transputer target architecture.

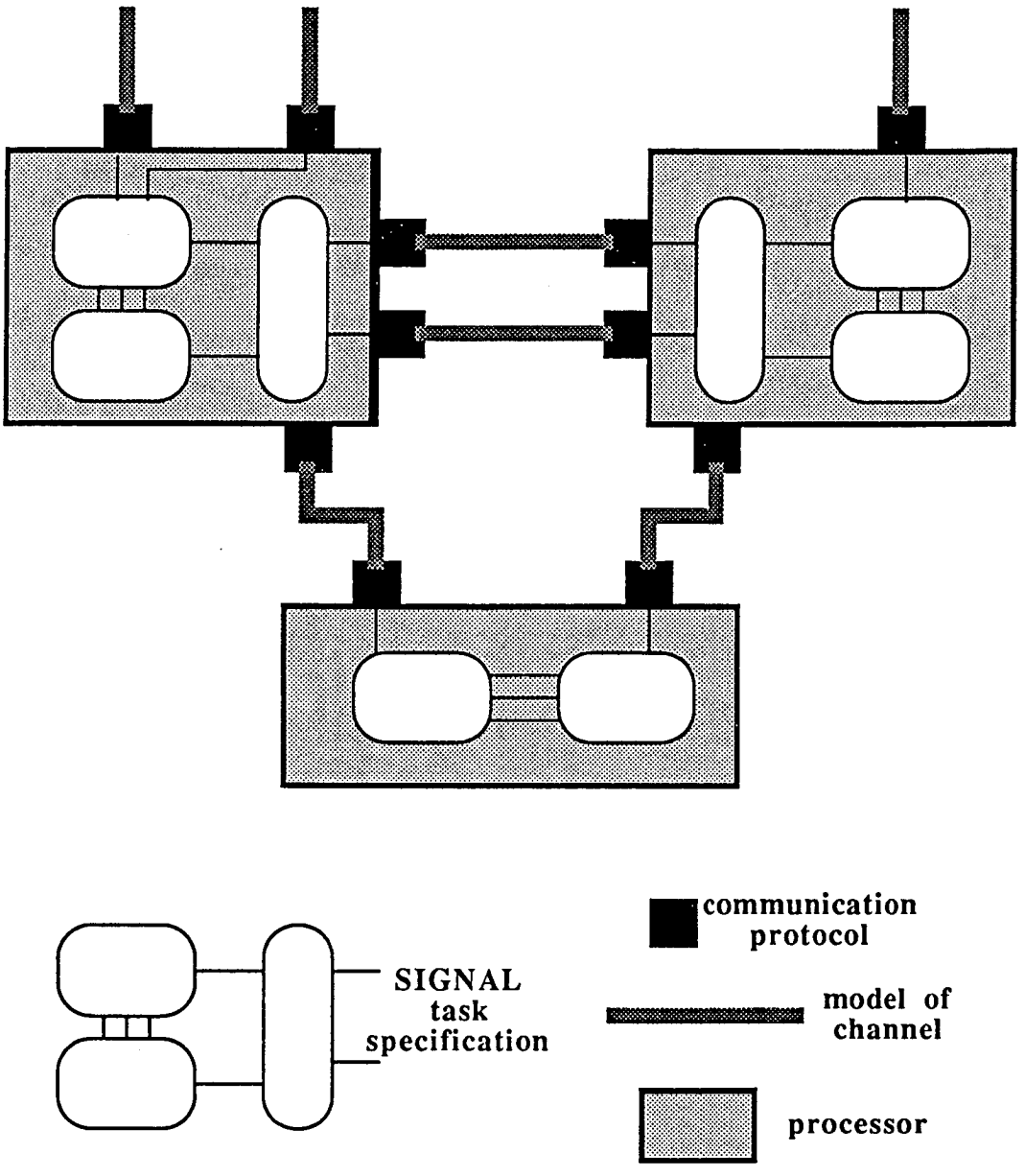

Figure 10: The Implementation Modelling Program.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 483 **POLYNOMIAL IDEAL THEORETIC METHODS IN DISCRETE EVENT, AND HYBRID DYNAMICAL SYSTEMS**
Michel LE BORGNE, Albert BENVENISTE, Paul LE GUERNIC,
20 Pages, Juillet 1989.

PI 484 **IMPLEMENTING ATOMIC RENDEZVOUS WITHIN A TRANSACTIONAL FRAMEWORK**
Jean-Pierre BANATRE, Michel BANATRE, Christine MORIN
22 Pages, Juillet 1989.

PI 485 **THE MAPPING OF LINEAR RECURRENCE EQUATIONS ON REGULAR ARRAYS**
Patrice QUINTON, Vincent VAN DONGEN
40 Pages, Juillet 1989.

PI 486 **SYNTHESIS OF A NEW SYSTOLIC ARCHITECTURE FOR THE ALGEBRAIC PATH PROBLEM**
Abdelhamid BENAINI, Patrice QUINTON, Yves ROBERT,
Yannick SAOUTER, Bernard TOURANCHEAU
34 Pages, Juillet 1989.

PI 487 **PLANS SIMULATION USING TEMPORAL LOGICS**
Eric RUTTEN, Lionel MARCE
40 Pages, Juillet 1989.

PI 488 **ON FINITE LOOPS IN LOGIC PROGRAMMING**
Philippe BESNARD
20 Pages, Septembre 1989.

PI 489 **LTA : UN LANGAGE DE TRAITEMENT D'ARBRES**
Dalila HATTAB
24 Pages, Septembre 1989.

PI 490 **THE SIGNAL SOFTWARE ENVIRONMENT FOR REAL-TIME SYSTEM SPECIFICATION, DESIGN, AND IMPLEMENTATION**
Albert BENVENISTE, Paul LE GUERNIC, Christian JACQUEMOT
34 Pages, Septembre 1989.

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