SIGNAL as a model for Real-Time and Hybrid Systems

Albert BENVENISTE, Michel LE BORGNE, Paul LE GUERNIC*

Abstract

Hybrid Systems are models of systems operating in real-time and handling events as well as "continuous" computations. The SIGNAL formalism for Hybrid Systems is presented in this extended abstract. Its expressive power is discussed, and a general method to associate various formal systems with it is presented and illustrated on deriving the present SIGNAL compiler.

1 Introduction: Real-Time and Hybrid Systems

It is commonly accepted to call *real-time* a program or system that receives external interrupts or reads sensors connected to physical world and outputs commands to it. As an example, let us discuss the case of an aircraft control system. Measurements are received from sensors and processed by the control loops to produces commands as outputs for the actuators: this involves various kinds of numerical computations. Switching from one operating mode to another one can be performed automatically or by the pilot: in both cases, events are received that control the various computations in some discrete event mode. For safety purposes, on-line failure detection and reconfiguration is performed by taking advantage of the redundancies in the aircraft system: actuators and sensor failure detection procedures are numerical computations that produce alarms and various detections which in turn result in reconfiguring the operating mode. From this follows that discrete events and computations are tightly coupled in a fairly symmetric way. Also, in aircraft systems, response times are often critical. They depend on the particular tasks being performed at the considered instant. On the other hand, timeouts due to timing constraints can influence the task being performed. To summarize, in real-time systems, both aspects of "what" should be implemented and "how fast" it should run must be considered. Similarly, aspects of "discrete control" and "continuous computations" deeply interact, and both interact also with response times via the mechanism of timeouts. The term of *Hybrid System (HS)* recently emerged

*A.B and P. L.G. are with INRIA-IRISA, M. L.B. is with IRISA-University, Campus Beaulieu, 35042 RENNES Cedex, FRANCE, name@irisa.fr. **Keywords:** real-time, semantics of programming languages, theory of parallel computation.

[3, 14] to refer to formalisms providing an attempt to cover the above mentioned issues in some unified way.

For instance, various kinds of transition systems are proposed in [10, 11] to combine qualitative and quantitative aspects of real-time. Similarly, [14] proposes to generalize a similar approach to Hybrid Systems. In both cases, the expressive power of the models is explicitly described in the axioms and inference rules of the semantics, so that expanding this expressive power is achieved by making the model more complicated. In this paper, we analyze an alternative formalism for HS that has beeen firstly proposed in [3, 2]. The main features of this formalism are:

1. it is both simple and "universal" as far as expressive power for specification is concerned: discrete events and computations are encompassed, and timing constraints can be specified; it does not rely on the notion of transition system, but handles "traces" or "behaviours", see [2];

2. it has various formal systems associated with it that can be used to verify or even synthesize some properties related to the above mentioned issues.

While "universal" expressive power is achieved realtively easily as we shall see later, only limited reasoning capability can be expected in turn, due to problems of undecidability. A major objective of this paper is to present a systematic approach to derive formal systems associated with our universal formalism, which concentrate on some particular property, e.g., discrete event features, timing constraints, etc...

2 Hybrid Systems and the SIGNAL language

A first major issue is that of the very nature of time for Hybrid Systems. Complex applications such as the one mentioned above are inherently distributed in nature. Hence every subsystem possesses its own time reference, namely the ordered collection of all the communications or actions this subsystem performs: in sensory based control systems, each sensor posses its own digital processing with proper sampling rate, actuators generally have a slower sampling rate than sensors, and moreover the software devoted to monitoring only reacts to various kinds of alarms that are triggered internally or externally. Hence the nature of time in Hybrid Systems is by no means universal, but rather local to each subsystem, and consequently multiform. This very fundamental remark justifies the kind of model for Hybrid System we use in this paper.

Our model handles infinite sequences of data with a certain kind of restricted asynchronism. Assume that each sequence, in addition to the normal values it takes in its range, can also take a special value representing the *absence* of data at that instant. The symbol used for absence is $\perp$. Therefore, an infinite time sequence of data (we shall refer to informally as a *signal* in this discussion) may look like

$$1, -4, \perp, \perp, 4, 2, \perp, ... \tag{1}$$

which is interpreted as the signal being absent at the instants $n = 3, 4, 7, ...$ etc. *Systems specified via constraints on signals of the form (1) will be termed Hybrid Systems (HS).*

A typical way of specifying such constraints will be to write equations relating different signals. The following questions are immediate from this definition:

(1) If a single signal is observed, should we distinguish the following samples from each other?

$$\{1,-4,\perp,\perp,4,2,\perp,...\}\ ,\ \{\perp,1,\perp,-4,\perp,4,\perp,2,\perp,...\}\ ,\ \{1,-4,4,2,...\}$$

Consider an "observer"[1] who monitors this single signal and does nothing else. Since he is assumed to observe only *present* values, there is no reason to distinguish the samples above. In fact, the symbol $\perp$ is simply a tool to specify the *relative* presence or absence of a signal, given an *environment,* i.e. other signals that are also observed. Jointly observed signals taking the value $\perp$ 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 present simultaneously. This notion of time makes no reference to any "physical" universal clock: time is rather local to each particular subset of signals in consideration.

(2) How to interconnect two Hybrid Systems? Consider the following two Hybrid Systems specified via equations:

$$y_n = \text{ if } x_n > 0 \text{ then } x_n \text{ else } \perp \tag{2}$$

and the usual addition on sequences, namely

$$z_n = y_n + u_n \tag{3}$$

In combining these HS, it is certainly preferable to match the successive occurrences $y_1, y_2, ...$ in (3) with the corresponding *present* occurrences in (2) so that the usual meaning of addition be met. But this is in contradiction with the bruteforce conjunction of equations (2,3)

$$\begin{aligned} y_n &= \text{ if } x_n > 0 \text{ then } x_n \text{ else } \perp \\ z_n &= y_n + u_n \end{aligned}$$

which yields $z_n = \perp + u_n$ whenever $x_n \leq 0$. In appendix A a denotational model for HS firstly introduced in [3] and improved in [2] is reported as a complementary information, it provides an adequate answer to the question of how to properly interconnect equations (2,3). This model is then used to establish the semantics of the SIGNAL language we introduce informally in the following section. To summarize, our formalism will provide a *multiform* but *coherent* notion of time. Other formalisms using the same approach to handle time are the so-called *synchronous languages* [1, 8, 9].

[1] in the common sense, no mathematical definition is referred to here

2.1 SIGNAL-kernel

We shall introduce only the primitives of the SIGNAL language, and drop any reference to typing, modular structure, and various declarations; the interested reader is referred to [13]. SIGNAL 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*. If x is a signal, we denote by $\{x_n\}_{n\geq 1}$ the sequence of its values when it is present. Signals that are always present simultaneously are said to have the same *clock*, so that clocks are equivalence classes of simultaneously present signals. Instructions of SIGNAL are intended to relate clocks as well as values of the various signals involved in a given system. We term a system of such relations *program*; programs may be used as modules and further combined as indicated later.

A basic principle in SIGNAL is that a single name is assigned to every signal, so that in the sequel, identical names refer to identical signals. The kernel-language SIGNAL possesses 5 instructions, the first of them being a generic one.

```
 (i)   R(x1,...,xp)
(ii)   y := x $1 init x0
(iii)  y := x when b
(iv)   y := u default v
 (v)   P | Q
(vi)    P !! x1,...,xp
```

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

$$\mathtt{R(x1,...,xp)} \iff \forall n:\ R(\mathtt{x1}_n,...,\mathtt{xp}_n) \text{ holds}$$

where $R(...)$ denotes a relation and the index n enumerates the instants at which the signals xi are present. Examples are functions such as z := x+y ($\forall n : \mathtt{z}_n = \mathtt{x}_n + \mathtt{y}_n$). A byproduct of this instruction is that *all referred signals must be present simultaneously, i.e. they must have the same clock.* This is a generic instruction, i.e. we assume a family of relations is available. If R(...) is the universal relation, i.e., it contains all the p-tuples of the relevant domains, the resulting SIGNAL instruction only constrains the involved signals to have the same clock: the so obtained instruction will be written x ^= y and only forces the listed signals to have the same clock.

(ii) shift register.

$$\mathtt{y := x\ \$1\ init\ x0} \iff \forall n > 1 : \mathtt{y}_n = \mathtt{x}_{n-1}, \mathtt{y}_1 = \mathtt{x0}$$

Here the index n refers to the values of the signals when they are *present*. Again this instruction forces the input and output signals to have the same clock.

(iii) condition (b is boolean): y equals x when the signal x and the boolean b are available and b is true; otherwise, y is absent; the result is an event-based undersampling

of signals. Here follows a diagram summarizing this instruction:

x :	1	2	⊥	⊥	3	4	⊥	⊥	5	6	9	...
b :	*t*	*f*	*t*	⊥	*f*	*t*	*f*	⊥	⊥	*f*	*t*	...
y :	1	⊥	⊥	⊥	⊥	4	⊥	⊥	⊥	⊥	9	...

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

u :	1	2	⊥	⊥	3	4	⊥	⊥	5	⊥	9	...
v :	⊥	⊥	⊥	3	4	10	⊥	8	9	2	⊥	...
y :	1	2	⊥	3	3	4	⊥	8	5	2	9	...

Instructions (i-iv) specify the elementary programs.

(v) combination of already defined programs: signals with common names in P and Q are considered as identical. For example

```
(| y := zy + a
 | zy := y $1 x0 |)
```

denotes the system of recurrent equations:

$$\begin{aligned} y_n &= zy_n + a_n \\ zy_n &= y_{n-1},\ zy_1 = \mathtt{x0} \end{aligned} \tag{4}$$

On the other hand, the program

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

yields

$$\text{if } x_n > 0 \text{ then } \begin{cases} y_n = x_n \\ z_n = y_n + u_n \end{cases} \quad \text{else } y_n = u_n = z_n = \perp \tag{5}$$

where (x_n) denotes the sequence of present values of x. Hence the communication | causes ⊥ to be inserted whenever needed in the second system z:=y+u. This is what we wanted for the example (2,3). Let us explain this mechanism more precisely. Denote by $u_1, u_2, u_3, u_4, \cdots$ the sequence of the present values of u (recall that y,z are present simultaneously with u). Then, according to point (1) of the discussion at the beginning of this section, $u_1, u_2, u_3, u_4, \cdots$ is equivalent to its following expanded version:

$$\mathtt{u}\ :\ \perp, u_1, \perp, \perp, u_2, u_3, \perp, \perp, \perp, u_4, \perp, \cdots,$$

for any finite amount of "⊥"s inserted between successive occurrences of u. Assuming all signals of integer type, suppose the following sequence of values is observed for x:

$$\mathtt{x}\ :\ -2, +1, -6, -4, +3, +8, -21, -7, -2, +5, -9, \cdots$$

Then the amount of inserted "⊥"s for the above expanded version of u turns out to fit exactly the negative occurrences of x: this flexibility in defining u allows us to match the present occurrences of u with the present occurrences of y, i.e., the positive occurrences of x. This mechanism is formalized in the model of the appendix.

(vi) restriction of program P to the mentioned list of signals: other signals involved in P are local and are not visible when communication is considered.

2.2 Some macros related to timing

An **event** type signal T (or "pure" signal) is an always *true* boolean signal. Hence "not T" denotes the boolean signal with clock T which always carries 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. 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". 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;

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

Finally, various kinds of *timers* will be useful, and some of them are listed below:

```
Y :=  X in ]S,T]          (i)
N := #X in ]S,T]         (ii)
```

The expression (i) delivers those present X's which occur within the left-open and right-closed interval]S,T], where S and T are both pure signals. Here follows a diagram showing the behaviour of this macro, we added the boolean signal "in]S,T]" which is delivered when X,S, or T is present, and is *true* within intervals]S,T] and *false* otherwise (this boolean is called BELONGS_TO_INTERVAL in the expansion of this macro we show below):

X:	1	2	⊥	⊥	3	4	1	⊥	5	6	9	...
S:	⊥	⊥	*t*	*t*	⊥	⊥	⊥	⊥	⊥	*t*	⊥	...
T:	*t*	⊥	⊥	⊥	⊥	*t*	⊥	⊥	⊥	⊥	*t*	...
in]S,T]:	*f*	*f*	*f*	*t*	*t*	*t*	*f*	⊥	*f*	*f*	*t*	...
Y:	⊥	⊥	⊥	⊥	3	4	⊥	⊥	⊥	⊥	9	...

Expression (ii) 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. Expanding these macros and their variations on the shape of the considered intervals ([S,T[, [S,T], etc...) into the primitive SIGNAL statements is easily done. For instance, (i) is defined via the following SIGNAL module:

```
(|(| IN_S_T ^= (S default T default (event X))
   |(| HITTING_S_T := (not T) default S default IN_S_T
     | IN_S_T := HITTING_S_T $1 init false |)|)
 | Y := X when IN_S_T |)
```

The hierarchy of submodules is depicted by the amount of |. This program is composed of two blocks. The meaning of the second one (last equation) is immediate, thus we concentrate on the first one which purpose is to produce the boolean IN_S_T (corresponding to "in]S,T]" discussed above). The first equation indicates when this signals has to be delivered. The block composed of the equations 2 and 3 delivers the value of IN_S_T: the boolean signal HITTING_S_T corresponds to "in [S,T[".

2.3 A few examples

The little programs implementing the formulae (4,5) illustrated how SIGNAL can be used to specify flows of computations and the emission of events that can result. In this subsection, we discuss other aspects of the language.

Specifying logical temporal properties: a single token buffer

Consider a memory with content M, which can be written (signal WRITE) and read (signal READ):

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

The first instruction expresses that the memory M is refreshed when WRITE is received, otherwise the previous value (M$1) is kept. Note that the clock of M is not entirely determined, it only has to be more frequent than that of WRITE. The second equation expresses that, when reading is wanted (event READ), it actually occurs and provides us with a READ signal carrying the value of M. Consequently, the clock of M has to be more frequent that that of READ. To proceed further on, let us encode the status (being written or being read) of the memory as follows:

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

Now suppose that writing in the memory is allowed only when the previous value of the memory has been read. This constraint is expressed by the following equation:

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

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

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

Finally, putting these three additional equations together specifies a single token buffer.

Specifying timing constraints

Assume now that the single token buffer is being used as a mailbox by some other module and it is desired that, when stored, a message must be read within some specified delay. A corresponding SIGNAL specification is as follows:

```
(| event READ ^< TICK
 | N := #TICK in ]WRITE,READ]
 | N < MAX_TIME |)
```

The first statement expresses that the mailbox can be checked at any `TICK` instant. The second equation counts the delay between writing and reading the message in terms of `TICK`s. Finally, the last statement expresses that the constraint (`N < MAX_TIME`) must be satisfied.

Basic problems

As the two above examples show, SIGNAL programs generally express *constraints* on the behaviors of their involved signals. This makes the composition of SIGNAL programs fairly obvious[2]. On the other hand, SIGNAL programs will generally attempt to specify real-time systems that are *transducers,* i.e., possess inputs that drive them and produce outputs. Hence implementing a SIGNAL specification consists in constructing a transducer producing all solutions to the considered system of SIGNAL equations. Getting a transducer form out of a SIGNAL specification written as constraints on signals requires a powerful compiler. This compiler must be able to "solve" the SIGNAL systems of equations in some way to transform them into some input/output map. So it has to be a sort of a "formal calculus system". One of the objectives of this paper is to explain informally how such formal calculi can be derived. By the way, other services are immediately provided such as proofs, since there is no distinction between properties to be checked (these are constraints) and programs on which properties must be checked (these are also constraints).

Discussion: expressive power and formal reasoning capability

As illustrated by the examples above, SIGNAL can be used to specify all key features we mentioned as being relevant to Hybrid Systems: computations, events and logic, timing constraints, and their mutual interaction. As far as the current SIGNAL compiler is concerned, the following should be noticed:

- the single token buffer specification involves only synchronization and logic and is thus fully handled by the SIGNAL compiler in the very same way as temporal logic does;

[2]this claim is also supported by the formal definition of the communication operator | in the appendix A which is very simple.

- in contrast, expressions such as "`X := Y+Z`" are handled via rewriting (each occurrence of the left hand-side can be replaced by the expression in the right hand-side), no formal property is handled about real signals, nor about their associated operations;
- finally, no formal calculus about quantitative time is performed by the current SIGNAL formal system, in particular it cannot be proved that performing two successive responses in less than $10\mu s$ results in an overall response time of less than $20\mu s$.

Hence the distinction between specifying and verifying must be emphasized: while the SIGNAL formalism has general expressive power [2], the SIGNAL formal system has limited (although quite powerful) capabilities. In [2, 3] we present a mathematical model for Hybrid Systems and use it to establish the semantics of SIGNAL. In particular it is shown in [2] that SIGNAL has maximum expressive power for Hybrid Systems description. In the next section we show ho to derive formal systems for reasoning about Hybrid Systems defined by SIGNAL.

3 Deriving formal calculi for Hybrid Systems

We first discuss the simple case of "pure" SIGNAL, i.e., of programs involving only synchronization and logic, i.e., `event` and `logical` data types. Then we discuss how this simple case can be generalized.

3.1 "Pure" SIGNAL: programs involving only synchronization and logic

Three labels are required to encode the status *absent, true, false*. The finite field $\mathcal{F}_3$ of integers modulo 3 is used for this purpose[3] via the coding:

$$\text{absent} \leftrightarrow 0,\ \text{true} \leftrightarrow +1,\ \text{false} \leftrightarrow -1$$

Using this coding, we define a mapping from syntactic SIGNAL expressions to equations in $\mathcal{F}_3$ (recall that all signals are of type `event, logical`). This mapping is shown in table 1. In this table, the first instruction is a sample of a SIGNAL instruction of type `(i)`, other ones are encoded similarly. In the coding of the second instruction, α denotes the current value of the internal state of the delay. Here and in the sequel, the generic notation x' denotes the next value of x. In particular, the coding of the instruction `B := A$1` involves two successive values of the state α; this equation expresses that, when a is received ($a^2 = 1$) it is fed into the next value α' of the state otherwise it is unchanged; then b receives the current state when a is received. Finally, clock(P) is the clock calculus of P and $\cup$ denotes conjunction.

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

SIGNAL equation	$\mathcal{F}_3$ coding (or "clock calculus")
`A or B = event A`	$a^2 = b^2\ ,\ ab(a-1)(b-1) = 0$
`B := A$1`	$\alpha' = (1-a^2)\alpha + a\ ,\ b = a^2\alpha$
`y := x when b`	$y = x(-b-b^2)$
`y := u default v`	$y = u + (1-u^2)v$
`P|Q`	clock(P) $\cup$ clock(Q)

Table 1: Encoding "pure" SIGNAL programs

Let us apply this technique to the mailbox example. To simplify the notations, we denote the various signals by their first letter, e.g., w for `WRITE`, etc... Applying the rules of table 1 to each successive instruction yields

$$\begin{array}{rrcl} (1) & \mu' & = & (1-m^2)\mu + m\ ,\ zm\ =\ m^2\mu \\ (1) & m & = & w + (1-w^2)zm \\ (2) & r & = & mr^2 \\ (3) & f & = & w^2 - (1-w^2)r^2 \\ \texttt{(FULL \$1)} & \xi' & = & (1-f^2)\xi + f\ ,\ zf = f^2\xi \\ (4) & w^2 & = & zf - f^2 \\ (5) & r^2 & = & -zf - f^2 \end{array} \tag{6}$$

A little algebra allows us to rewrite (6) as follows (comments are written for each equation):

$$\begin{array}{rcll} m^2 & = & f^2 = 1 & \text{fastest clock, always present by convention} \\ f' & = & -f & \texttt{FULL} \text{ is a flip-flop} \\ zm' & = & m & \\ w^2 & = & -f-1 & \texttt{WRITE} \Leftrightarrow \text{ box full} \\ r^2 & = & f-1 & \texttt{READ} \Leftrightarrow \text{ box empty} \\ r & = & mr^2 & \\ m & = & w + (1-w^2)zm & \texttt{M} \text{ stores the written value} \end{array} \tag{7}$$

While (6) was given as a fixpoint equation, i.e., in an implicit form, (7) is in explicit form: reading the equations from top to down yields an execution mode. This explicit form reveals that the single-token is *not* a transducer: we cannot consider that `WRITE` acts as an input and `READ` is the output. Instead, `FULL` acting as a flip-flop drives the synchronization, and the additional input is the *value* carried by `WRITE`, not its clock. We say that (7) is a *solved form* of (6). Deriving (7) from (6) amounts to applying elimination techniques to polynomial functions over $\mathcal{F}_3$. When the specified system actually was equivalent to a transducer, calculating the solved form provides us with this equivalent transducer. A very efficient version of this technique has been developed for the SIGNAL compiler [13, 7].

3.2 Developing formal calculi for general Hybrid Systems

Consider the following SIGNAL program:

input	R:	3	$\perp$	$\perp$	4	$\perp$	$\perp$	$\perp$	1	5	$\perp$	$\perp$	$\perp$	$\perp$	etc...
output	N:	3	2	1	4	3	2	1	1	5	4	3	2	1	etc...

Table 2: UPSAMPLING in SIGNAL

```
(| N := R default (ZN-1)
 | ZN := N$1
 | R ^= when (ZN=1) |)
```

In this program, R is assumed to be a strictly positive integer signal, and ZN has initial value 1. The behaviour of this program is depicted in table 2. This program serves as a basic mechanism for data dependent *upsampling* of the input signal R. It is a particular and powerful feature of the SIGNAL formalism that programs with upsampling can be specified, see [2] for an extensive discussion about this aspect. Two domains are encountered, namely the positive integers R, N, ZN, and the boolean (ZN=1).

Hence for general HS or SIGNAL programs the situation is drastically more difficult: infinite domains are involved such as integers, reals, etc... So the systems of equations corresponding to general SIGNAL programs will be *approximately* solved using a technique we shall describe now informally. We describe now how the above UPSAMPLING program is actually handled by the current SIGNAL compiler. While most of the formalisms for real-time or Hybrid Systems hide computations inside actions that are viewed as black-boxes [14, 10, 11], we refuse to do so, since such actions turn out to influence the real-time behaviour (comparing a real signal to a threshold can be a mechanism to produce interruptions). The following general procedure is proposed to overcome this difficulty:

1. Select the domains for which you want to provide a formal system; we already discussed this issue in the two preceding sections.

2. Equations involving other domains can be handled in the weakest way, namely via syntax-based rewriting; syntax-based rewriting algorithms amount to handling directed graphs; but these graphs vary dynamically according to the clock of the considered instant, we call them *dynamical graphs.*

The interested reader is referred to [3, 4] for details and formal definitions of dynamical graphs, we concentrate here on an informal discussion of the UPSAMPLING example. The coding of the three instructions of this program is given in table 3 (again we denote by B the boolean expression (ZN=1)). The notation $x \rightarrow y$ means that y depends on x when both are present, i.e., $x^2y^2 = 1$. Similarly, the notation $x \xrightarrow{h^2} y$ means that y depends on x when $x^2y^2h^2 = 1$ holds; this latter notation is for instance used in the coding of the **default** instruction. Some comments follow. Capitals are used (e.g., N) to refer to values of present signals, while variables of the clock calculus are written in lower cases. Values of present signals depend on their clock (e.g., $n^2 \rightarrow$ N). The second line of the conditional dependency graph is the coding of the **default**: the double dependency R $\rightarrow$ N $\xleftarrow{1-r^2}$ ZN is that between values, and the second one $n^2r^2 \rightarrow$ N expresses that,

clock calculus	conditional dependency graph
$n^2 = r^2 + (1-r^2)zn^2$	$n^2 \rightarrow \mathtt{N}, r^2 \rightarrow \mathtt{R}, zn^2 \rightarrow \mathtt{ZN},$ $\mathtt{R} \rightarrow \mathtt{N} \xleftarrow{1-r^2} \mathtt{ZN}, n^2r^2 \rightarrow \mathtt{N}$
$zn^2 = n^2$	
$b^2 = zn^2$ $r^2 = -b - b^2$	$b^2 \rightarrow b, \mathtt{ZN} \rightarrow b$

Table 3: Encoding the UPSAMPLING program

to compute the output N it is needed to know the clock deciding which dependency will be in force at the considered instant.

This coding is hybrid in nature: two different algebras are involved, namely the algebra of polynomial expressions in $\mathcal{F}_3$ variables, and labelled directed graphs. These graphs specify evaluation schemes for run time. Based on the actual syntactic form of the clock calculus, it is also possible to associate a graph with the clock calculus. Performing this for the UPSAMPLING program yields

$$r^2 \rightarrow n^2 \xleftarrow{1-r^2} zn^2 \ (= n^2 = b^2)$$
$$b \rightarrow r^2$$

Combining this graph with the preceding one yields for instance the circuit $b \rightarrow r^2 \rightarrow n^2 = b^2 \rightarrow b$ which has a clock equal to $b^2r^2 = r^2 \neq 0$. Thus combining these two graphs does not provide a partial order, so no evaluation scheme is derived at this point.

To overcome this *we use the calculus of $\mathcal{F}_3$ to replace the present equations of the clock calculus by other ones that are equivalent, in such a way that the combined resulting graphs be gobally circuitfree.* For instance, we can replace the clock calculus of the UPSAMPLING program by its solved form

$$zn^2 = n^2 = b^2 \ , \ r^2 = -b - b^2$$

which yields the graph with a single branch $b \rightarrow r^2$. The resulting global graph is finally (arrows without input nodes denote inputs of the evaluation scheme):

$$\begin{aligned}
\text{(input clock)} \qquad & \rightarrow b^2 = n^2 = zn^2 \rightarrow b \ , \ b^2 \rightarrow \mathtt{ZN} \\
\text{(input: memory content)} \qquad & \rightarrow \mathtt{ZN} \rightarrow b \rightarrow r^2 \\
\text{(input: value of R)} \qquad & r^2 \rightarrow \mathtt{R} \ , \quad \rightarrow \mathtt{R} \\
& \mathtt{ZN} \xrightarrow{1-r^2} \mathtt{N} \ , \ b^2 \rightarrow \mathtt{N} \ , \ r^2 \rightarrow \mathtt{N} \ , \ \mathtt{R} \rightarrow \mathtt{N}
\end{aligned}$$

No circuit is exhibited, so that a partial order can be associated with this graph. It is interesting to note that the input clock is that of the output N, and that only the value carried by R is an input: UPSAMPLING runs according to the demand driven mode.

Again, the method we presented here informally can be extended to other algebras. The current SIGNAL compiler is powered with a very fast implementation of the above procedure, see [13, 7]. Finally, the reader is referred to [4] for a formal presentation of the SIGNAL formal system in its present form.

3.3 Extension to other domains: quantitative real-time

We consider again the UPSAMPLING program, but we assume now that its input R is a positive *bounded* integer signal. Since all signals have now finite domains nothing really new happens compared to the elementary case of "pure" SIGNAL, for instance we may translate bounded integers into vectors of booleans. However we find it more convenient to use codings that are tightly tailored to each of these domains, since the resulting coding will generally be more compact, thus memory saving and efficient calculations should result; in particular, the very efficient formal calculi developed for $\mathcal{F}_3$ [13, 7] generalize to any $\mathcal{F}_p$.

Thus the boolean signal (ZN=1) is encoded using $\mathcal{F}_3$ as before. Similarly, the finite field $\mathcal{F}_p$ of integers modulo p can be used with p large enough to encode the integer signals R, N, ZN with the following mapping:

$$\text{absent} \leftrightarrow 0,\ 1 \leftrightarrow 1, \cdots,\ p-1 \leftrightarrow p-1$$

In this coding, clocks of integer signals are recovered as follows

$$\text{R absent} \leftrightarrow r^{p-1} = 0 \quad , \quad \text{R present} \leftrightarrow r^{p-1} = 1$$

From these remarks, the coding of the UPSAMPLING program follows (we denote by B the boolean signal (ZN=1):

$$\begin{array}{rcll} n & = & r + (1 - r^{p-1})(zn - 1) & \text{(i) first instruction} \\ & & & \\ \nu' & = & (1 - n^{p-1})\nu + n & \\ zn & = & n^{p-1}\nu & \text{(ii) second instruction} \\ & & & \\ b & = & zn^{p-1}(1 + (zn-1)^{p-1}) & \text{(iii) B as a function of ZN} \\ r^{p-1} & = & -r - r^2 & \text{(iv) last instruction} \end{array} \tag{8}$$

However in doing this, a new difficulty appears. Equations (i,ii) are polynomial equations within $\mathcal{F}_p$ and thus can be handled in a way similar to that of pure SIGNAL programs. Unfortunately, equation (iii) is a function mapping some $\mathcal{F}_p$-expression into $\mathcal{F}_3$; similarly, equation (iv) is a function mapping some $\mathcal{F}_3$-expression into $\mathcal{F}_p$. This is again a hybrid coding similar to that of the preceding subsection. We can handle it in two different ways.

1. Since B,ZN,R only are involved in the "hybrid" equations (iii,iv), we may first eliminate ν and N from equations (i,ii). This is equivalent to projecting the dynamical system (i,ii) onto the components (zn, r) only, and is generally performed using elimination techniques in $\mathcal{F}_p$ via efficient algorithms as mentioned before. In our case this yields the unique equation

   $$zn' = r + (1 - r^{p-1})(zn - 1)$$

   which we handle in combination with (iii,iv) via exhaustive scanning. In this way of doing, efficient algorithms can be called for to project each homogeneous

subsystem (here (i,ii)) onto its interfaces to other homogeneous ones (here `B,ZN,R`). Then the problem reduces to that of standard model checking techniques on the joint behaviour of the set of all such interfaces.

2. Perform first as before the reduction to the interfaces of the homogeneous subsystems. The remaining "hybrid" equations are then approximately solved using the graph method of the preceding subsection. This is a less powerful but likely faster method of compilation.

As sketched on this example, formal calculi can be developed for properties relevant to synchronization, logic, and quantitative timing. In particular, (qualitative and quantitative) *real-time specifications for discrete event systems can be synthesized.*

4 Conclusion

We have presented the SIGNAL formalism to specify Hybrid Systems. SIGNAL is currently used to specify and program real-time systems [13] according to the principles of synchrony [1].

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 from TNI Inc., Brest, France. SILDEX[4], a X-windows based graphical environment is provided for both program editing and on-line monitoring and supervision of the execution. C, FORTRAN, or ADA code is produced. Experiments have been performed based on this version to produce distributed OCCAM 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, an aircraft control system, were the major ones.

We have discussed the expressive power of SIGNAL and have illustrated its generality, formal studies are also available in [2] to support this claim.

We have presented an original and general method for deriving formal calculi for Hybrid Systems. The central notion of this method is that of *dynamical graph* and is used in the current SIGNAL compiler to handle synchronization, logic, and data dependencies.

Several directions for future research are currently pursued. Improving the efficiency and power of the formal system that handles $\mathcal{F}_3$–based dynamical graphs is a key issue to fast and efficient compilation [12]. Deriving efficient systems to handle bounded integers will open the route to quantitative real-time: a major issue is to handle the tradeoff efficiency/generality of such formal calculi. Finally, as lengthly discussed in [2], the SIGNAL formalism is already very close to models of stochastic processes: adding a *single* instruction to SIGNAL provided us with the SIGN*alea* extension [5]. SIGN*alea*

[4]SILDEX is a TradeMark of TNI.

is able to specify and handle various probabilistic real-time systems such as queuing networks or uncertain real-time information processing systems.

Appendix

A A *trace* model for HS and a semantics of SIGNAL

In this appendix, a mathematical model for HS is presented, and used to formally define SIGNAL. The reader is referred to Section 2 for the motivation of the following definitions.

A.1 Histories, signals, clocks

Consider an alphabet (finite set) A of typed variables called *ports*. For each $a \in A$, $\mathcal{D}_a$ is the domain of values (integers, reals, booleans...) that may be carried by a at every instant. Introduce

$$\mathcal{D}_A = \bigcup_{a \in A} (\mathcal{D}_a \cup \{\perp\})$$

where the additional symbol $\perp$ denotes the absence of the value associated with a port at a given instant. For two sets A and B, the notation $A \rightarrow B$ will denote the set of all maps defined from A into B. Using this notation, we introduce the following objects.

Events. Events specify the values carried by a set of ports at a considered instant. The set of the A-events (or "events" for short when no confusion is likely to occur) is defined as

$$\mathcal{E}_A = A \rightarrow \mathcal{D}_A$$

Events will be generally denoted by ϵ and their domain by $\mathcal{D}(\epsilon)$. We shall denote by $\perp$ the "silent" event ϵ such that $\epsilon(a) = \perp\ \forall a \in \mathcal{D}(\epsilon)$.

Traces. Traces are infinite sequences of events. Let $\mathbf{N}_+ = \{1, 2, ...\}$ denote the set of integers, then the set of *A-traces* (or simply "traces") is defined as

$$\Theta_A = \mathbf{N}_+ \rightarrow \mathcal{E}_A$$

Compressions. The *compression* of an A-trace T (deleting the silent events) is defined as the (unique) A-trace S such that:

$$S_n = T_{k_n}$$

where

$$k_0 = \min\{m \geq 0 :\ T_m \neq \perp\},\ k_n = \min\{m > k_{n-1} :\ T_m \neq \perp\}$$

where $\min \emptyset = +\infty$ by convention. The compression of a trace T will be denoted by $T\downarrow$.

Histories and signals. The condition

$$T\downarrow \;=\; T'\downarrow$$

defines an equivalence relation on traces we shall denote by $T \sim T'$. The corresponding equivalence classes are called *histories*. The set of all possible histories on A will be denoted by Ω_A, so that we have[5]

$$\Omega_A = (\Theta_A)_{/\sim}$$

Elements of Ω_A will be generically denoted by ω_A or simply ω when no confusion can occur. While the notion of trace refers to a particular environment (since the $\perp$'s are explicitly listed), the notion of history does not. Since

$$\Omega_A = [\mathbf{N}_+ \to (A \to \mathcal{D}_A)]_{/\sim}$$

any $\omega_A \in \Omega_A$ may be written as

$$\omega_A = (\omega_a)_{a \in A} \tag{9}$$

and the ω_a's are termed *signals*. Hence a signal is a component of a history specified by selecting a particular port in the alphabet A. The notion of "signal" has been informally discussed in section 2-(1), where we motivated the definition of signals and histories as equivalence classes with respect to the relation $\sim$.

Clocks. Extend the domains $\mathcal{D}_a$ with another distinguished value $\top$, intended to encode the status "present" regardless of any particular value. Consider the map $chronos_{\mathcal{D}} \in \mathcal{D}_A \to \{\perp, \top\}$ defined by

$$chronos_{\mathcal{D}}(\perp) = \perp,\ chronos_{\mathcal{D}}(x) = \top \text{ for } x \neq \perp$$

For each event $\epsilon \in \mathcal{E}_A$, there is a unique map in $\mathcal{E}_A \to \mathcal{E}_A$ making the following diagram commutative, denote it by $chronos_{\mathcal{E}}$:

$$\begin{array}{ccc} & A & \\ \epsilon \swarrow & & \searrow\ chronos_{\mathcal{E}}(\epsilon) \\ \mathcal{D}_A & \xrightarrow{chronos_{\mathcal{D}}} & \mathcal{D}_A \end{array}$$

Similarly, there is a unique map in $\Theta_A \to \Theta_A$, we denote by $chronos_\Theta$, making the following diagram commutative

$$\begin{array}{ccc} & \mathbf{N}_+ & \\ T \swarrow & & \searrow\ chronos_\Theta(T) \\ \mathcal{E}_A & \xrightarrow{chronos_{\mathcal{E}}} & \mathcal{E}_A \end{array}$$

This map satisfies the condition $T_1 \sim T_2 \Rightarrow chronos_\Theta(T_1) \sim chronos_\Theta(T_2)$, so that it induces a map in $\Omega_A \to \Omega_A$ we shall now denote by *chronos*: the *chronos* of a history is another history which summarizes the status {present/absent} of each of its signals (i.e. components).

Now, given $\omega \in \Omega_A$ and $a \in A$, consider the signal of port a of the history $chronos(\omega)$: this signal summarizes the *relative* status present/absent of the signal ω_a given the other signals involved in the history ω. We shall call this signal the *clock* of ω_a, or the clock of a for short when no confusion is likely to occur, and denote it by $clock(\omega_a)$ or $clock(a)$.

[5] $\cdot_{/\sim}$ denotes here the quotient space by the relation $\sim$

A.2 HS

Definition of HS. A HS is simply a subset

$$\Omega \subset \Omega_A$$

of the set of all histories on A. In other words, we consider a SIGNAL program, as a way to specify "legal" histories.

Restricting HS. Consider a subset A' of the alphabet A. The inclusion $A' \subset A$ induces a projection from $\mathcal{E}_A$ onto $\mathcal{E}_{A'}$ we denote by $\epsilon \longrightarrow \epsilon_{!!A'}$. Following the same argument as for the definition of clocks, we derive the following family of *restrictions* we generically denote by $\cdot_{!!A'}$. First, the following commutative diagram

$$\begin{array}{ccc} & \mathbf{N}_+ & \\ T \swarrow & & \searrow T_{!!A'} \\ \mathcal{E}_A & \xrightarrow{\cdot_{!!A'}} & \mathcal{E}_{A'} \end{array}$$

uniquely defines the restriction $T \longrightarrow T_{!!A'}$ on traces. Since $T_1 \sim T_2 \Rightarrow (T_1)_{!!A'} \sim (T_2)_{!!A'}$ holds, a restriction on histories $\omega \longrightarrow \omega_{!!A'}$ may be defined, which finally yields a restriction on HS we denote by

$$\Omega \longrightarrow \Omega_{!!A'}$$

This restriction maps the set of HS defined over the alphabet A onto the set of HS defined over the alphabet A'. The HS $\Omega_{!!A'}$ is called the *restriction* of Ω to (the subalphabet) A': only the signals with ports in A' are visible from outside and may be used for HS communication we shall define next.

HS communication. Consider two HS Ω_1, Ω_2 respectively defined over the alphabets A_1 and A_2. Set $A = A_1 \cup A_2$. Then $\Omega_1 | \Omega_2$ will denote the maximal[6] HS Ω defined over the alphabet A satisfying the following conditions:

$$\begin{aligned} \Omega_{!!A_1} &\subseteq \Omega_1 \\ \Omega_{!!A_2} &\subseteq \Omega_2 \end{aligned}$$

In other words, the communication constrains the signals in Ω_1 and Ω_2 of shared port to be identical (i.e. to be present simultaneously and then carry the same value). This is exactly what we wanted while discussing the example of eqns (2,3).

A.3 The definition of SIGNAL

According to the preceding section, in order to specify an HS over a given alphabet, we have to describe a subset of all histories that can be built upon this alphabet. Since histories are defined as equivalence classes of traces with respect to the relation $\sim$, this may be done by *listing a family of constraints on the set of all traces* that can be built on this alphabet. The equivalence classes of the so specified traces are the specified histories. This is what we shall do next.

[6] with respect to the order by inclusion $\Omega' \subseteq \Omega$ defined on HS

Instruction (i): `R(x1,...xp)`

$$\begin{aligned} \forall n \in \mathbf{N}_+,\ \forall \mathtt{i} \ &: \ \mathtt{xi}_n \neq \perp \\ \forall n \in \mathbf{N}_+ \ &: \ R(\mathtt{x1}_n, ..., \mathtt{xp}_n) \text{ holds} \end{aligned}$$

Here, the notation $\mathtt{xi}_n$ denotes the value carried by the port with name `xi` at the n-th instant of the considered trace. This notation will be further used in the sequel of this subsection.

Instruction (ii): `y := x $1 x0`

$$\begin{aligned} \forall n \in \mathbf{N}_+ \ &: \ \mathtt{x}_n \neq \perp \\ \forall n > 1 \ &: \ \mathtt{y}_n = \mathtt{x}_{n-1} \\ &\quad\ \mathtt{y}_1 = \mathtt{x0} \end{aligned}$$

Instruction (iii): `y := x when b`

$$\forall n \in \mathbf{N}_+,\ \mathtt{y}_n = \begin{cases} \text{if } \mathtt{x}_n \neq \perp \text{ and } \mathtt{b}_n = \mathit{true} \text{ then } \mathtt{x}_n \\ \text{else } \perp \end{cases}$$

Instruction (iv): `y := u default v`

$$\forall n \in \mathbf{N}_+,\ \mathtt{y}_n = \begin{cases} \text{if } \mathtt{u}_n \neq \perp \text{ then } \mathtt{u}_n \\ \text{else if } \mathtt{u}_n = \perp \text{ and } \mathtt{v}_n \neq \perp \text{ then } \mathtt{v}_n \\ \text{else } \perp \end{cases}$$

Instruction (v): `P | Q`

We already defined the operator | on HS.

Instruction (vi): `P !! x1,...,xp`

We already defined the restriction on HS.

ACKNOWLEDGEMENT: the authors gratefully acknowledge Oded Maler and an anonymus reviewer for constructive criticism and fruitful remarks on preliminary versions of this paper.

References

[1] A. BENVENISTE, G. BERRY, "Real-Time systems design and programming", *Another look at real-time programming*, special section of *Proc. of the IEEE,* to appear Sept. 1991.

[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, to appear in *Science of Computer Programming.*

[5] A. BENVENISTE, "Constructive probability and the SIGN*alea* language", IRISA res. rep., 1991.

[6] B. BUCHBERGER, "Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory" N.K. Bose (ed.), Multidimensional Systems Theory, 184-232, D. Reidel Publishing Company.

[7] L. BESNARD, Thesis, IFSIC-IRISA, 1991.

[8] F. BOUSSINOT, R. DE SIMONE, "The ESTEREL language", *Another look at real-time programming,* special section of *Proc. of the IEEE,* to appear Sept. 1991.

[9] N. HALBWACHS, P. CASPI, D. PILAUD, "The synchronous dataflow programming language LUSTRE", *Another look at real-time programming,* special section of *Proc. of the IEEE,* to appear Sept. 1991.

[10] T.A. HENZINGER, Z. MANNA, A. PNUELI, "An Interleaving Model for Real-time", Jersalem Conf. on Information Technology 1990, IEEE Computer Society Press.

[11] T.A. HENZINGER, Z. MANNA, A. PNUELI, "Temporal proof methodologies for Real-time systems", POPL'91.

[12] 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.

[13] P. LE GUERNIC, T. GAUTIER, M. LE BORGNE, C. LE MAIRE, "Programming real-time applications with SIGNAL", *Another look at real-time programming,* special section of *Proc. of the IEEE,* to appear Sept. 1991.

[14] X. NICOLLIN, J. SIFAKIS, S. YOVINE, "From ATP to Timed Graphs and Hybrid Systems", REX workshop "Real-Time, theory in practice", Mook, The Netherlands, June 3-7, 1991.