



## Sequencing date flow tasks in SIGNAL

Eric Rutten, Paul Le Guernic

### ► To cite this version:

Eric Rutten, Paul Le Guernic. Sequencing date flow tasks in SIGNAL. [Research Report] RR-2120, INRIA. 1993. inria-00074552

**HAL Id: inria-00074552**

**<https://inria.hal.science/inria-00074552>**

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.

***Sequencing data flow tasks in SIGNAL***

Eric Rutten, Paul Le Guernic

**N° 2120**

Novembre 1993

PROGRAMME 2

Calcul symbolique,  
programmation  
et génie logiciel ***rapport  
de recherche*****1993**





# Sequencing data flow tasks in SIGNAL

Eric Rutten\*, Paul Le Guernic\*\*

Programme 2 — Calcul symbolique, programmation et génie logiciel  
Projet EP-ATR

Rapport de recherche n° 2120 — Novembre 1993 — 49 pages

**Abstract:** The SIGNAL language is a real-time, synchronized data-flow language. Its model of time is based on instants, and its actions are considered instantaneous. Various application domains such as signal processing and robotics require the possibility of specifying behaviors composed of successions of different modes of interaction with their environment. To this purpose, we introduce the notion of *time interval*, defined by a start and an end event, and denoting the series of its occurrences. Associating a time interval to a data-flow process specifies a *task* i.e., a non-instantaneous activity and its execution interval. Different ways of sequencing such tasks are described. We propose these basic elements at the programming language level, in the perspective of extensions to SIGNAL. Application domains feature the discrete sequencing of continuous, data-flow tasks, as is the case, for example, of robotic tasks.

**Key-words:** Data-flow tasks, task sequencing, real-time, time intervals.

(Résumé : *tsvp*)

**Acknowledgements:** Thanks to Thierry Gautier, Mohammed Belhadj, Olivier Maféïs, Philippe Le Parc and Sylvie Thiébaux for commenting on previous versions of this paper.

\*rutten@irisa.fr

\*\*leguernic@irisa.fr

Unité de recherche INRIA Rennes

IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)

Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

# Séquencement de tâches flot de données en SIGNAL

**Résumé :** Le langage SIGNAL est un langage temps-réel à flots de données synchronisés. Son modèle du temps est fondé sur les instants, et les actions y sont instantanées. Divers domaines d'application, tels que le traitement de signal ou le contrôle de processus physiques, requièrent la possibilité de spécifier des comportements composés de l'enchaînement de différents modes d'interaction avec leur environnement. À cette fin, nous introduisons la notion d'*intervalle de temps*, défini par des instants de début et de fin, et désignant la suite de ses occurrences. Associer un intervalle de temps à un processus flot de données spécifie une *tâche*, c'est-à-dire une activité non-instantanée et son intervalle d'exécution. Diverses manières de séquencer de telles tâches sont décrites. Nous proposons ces éléments de base au niveau du langage de programmation, en perspective de l'extension de SIGNAL. Les domaines d'applications comprennent le séquencement discret de tâches continues à flot de données, comme c'est le cas, par exemple, en robotique.

**Mots-clé :** Tâches flot de données, séquencement de tâches, temps-réel, intervalles de temps.

# 1 Motivation

## 1.1 Problem addressed

The problem we address is to provide ways of representing behaviors switching between different modes of continuous interaction with their environment. With regard to the environment, these behaviors consist in a succession of different functions relating outputs and inputs, each between discrete start and end events delimiting a time interval.

Application domains concerned are the control of physical, continuous processes in general, and in particular: robotics. Such applications more specifically feature the mixing of numerical computations on sensor data, and transitions along an automaton or place-transition graph. Example are:

- a speech recognition system [18]: the processing of the acoustic signal features a segmentation treatment. Boundaries of the segments are determined by changes that are detected by comparison to an average value computed on a time window on the past values of the signal, as illustrated in fig. 1. Establishing the first value of this average requires time before computation of the variance can be performed.

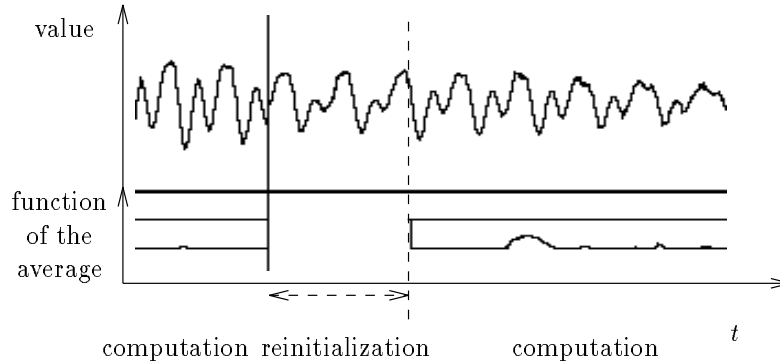


Figure 1: Phases in the segmentation of an acoustic speech signal.

Hence such an application presents successive phases: for each segment, phases of reinitialization alternate with the regular processing task performing the computation. Each of them is active on a time interval, in such a way that the end of the first one corresponds to the start of

the second one. The sequencing between these phases is intrinsic, in the sense that it is imposed by dependencies on results produced and consumed.

We want to provide a programmer with means to designate the phases in a processing by subdividing its activity interval into different modes, and to associate sub-activities to them.

- the sequencing of robotic tasks. A robotic task consists in performing a task function computing the command to the actuator from sensor input data. Task-level programming of a robot involves associating such activities with modes on which they are enabled. For example, movements toward some point can alternate with prehension tasks, or assembly of objects [10]. Each such task involves the performance of a specific task function conditioned by the satisfaction of conditions to be checked in the environment, for its start as well as for its end. For example, in an assembly application, a movement of the gripper towards an object to be grasped is an activity for which termination is defined by certain conditions on their proximity and orientation. Grasping itself involves another kind of control rule e.g., with precision movements and contact sensing. The transition between the two execution modes is driven by the reception of externally sensed information.

Reactivity plays an important role here, in that tasks can be performed in response to some situation or property of the environment. In other words, these modes correspond to the time interval on which the property holds.

We want to have operations allowing for the specification of sequencings of activities, relating their time interval to the state of the external world or to other intervals.

- the control of a digital watch, as a well-known example of hierarchical, parallel automata [14], that we treat in section 5.2. This example consists in associating to each of various modes (watch mode, different setting modes, stopwatch mode) a different activity. They are characterized by the way input from the buttons of the watch is being interpreted, and how and which internal values are being updated accordingly. This is a characteristic example of extrinsic sequencing: the different

modes of activity are independent, and the order in which they are sequenced is arbitrary.

However, the use of automata for the specification of such behaviors forces the order in which activities are performed, regardless of whether there exist constraints or dependencies necessitating this. For example, the order in which the various components of time (hours, minutes, seconds, day, month year, ...) are updated is fixed in the automaton.

We want to find more declarative ways to specify constraints between intervals on which activities are performed. For example here, they might have to be exclusive because of the limited number of buttons for controlling the digital watch.

The work presented here more precisely aims at providing structures supporting:

- description the attachment of activities to different modes,
- description of the succession of such modes.

The motivation for having both paradigms present in a hybrid language is to combine the advantages of on the one hand data flow languages for the specification of input/output relations in terms of equations or networks of operators, and on the other hand the association of such activities to modes or time intervals (on which their execution is enabled) for the sequencing of non-instantaneous tasks.

## 1.2 Context

The context of this work is reactive programming [20] and especially the synchronous approach to it and the languages adopting it: (ESTEREL, LUSTRE, SIGNAL, ARGOS, STATECHARTS, ...) have been developed for the safe design of real-time, reactive applications [3, 14]. Their formal definitions support their programming environments: compilers featuring static analysis, code generation, proof tools and simulation of dynamic behaviors. Their model of time supports reasoning about order and simultaneity of instants. Actions are modelled as being synchronous, meaning that their input and their result are present at the same instant.



From the point of view of their design as programming languages, they can be classified into two styles:

- imperative languages, textual such as `ESTEREL`, or graphical such as `STATECHARTS` and `ARGOS` or also `GRAFCET`<sup>1</sup> (even though its semantics does not explicitly adopt full synchrony); they are particularly well suited to the description of automata and the specification of sequencings (e.g., discrete controllers),
- synchronized data flow languages which are declarative (or equational), such as `LUSTRE` (which allows to describe functions) and `SIGNAL` (allowing to describe relations); they are best suited to the specification of data flow computations (e.g., signal processing or digital circuits).

As can be seen from the overview of related work in section 7, the sequencing and data flow languages do not generally feature constructs combining both paradigms, particularly structures enabling the simple sequencing of data flow tasks inside the synchronous framework.

### 1.3 Proposed approach

The solution proposed here is two-fold: it consists in

- explicitly defining modes (as a series of instants characterized by some property) in data flow processing, hence the intervals and the tasks,
- using time intervals for the specification of sequencings of such modes, hence the operations on intervals.

Associating a time interval to a data flow process specifies a *task* i.e., a non-instantaneous activity and its execution interval (start and end times). When entering an execution interval occurrence, a task can be started at its current state, as saved when it was suspended, or restarted in its initial state, if it was interrupted.

The intervals allow an explicit reference to states and transitions, supporting the specification of automata or place-transition networks. Hence, it is possible to specify hierarchical, parallel automata, controlling the sequencing of data flow tasks, that can themselves involve a sequencing of sub-tasks.

---

<sup>1</sup>Also known as `SEQUENTIAL FUNCTION CHARTS`.

However, sequencing is not necessarily imperative, in that executing a task does not necessarily directly depend on the action immediately preceding it. It depends on the satisfaction of some properties in its environment, for its starting as well as for its terminating (e.g., in the examples of section 1.1: associating specific actions to the initialization phase, when in precision movement use a certain sensor control procedure, ...). Hence the temporal arrangement of such tasks concerns less the actual sequencing of actions than relations between remanent properties in the environment, that can be described by states (or state variables). When the concerned modes do not involve complex interrelationships, their description is quite simple in the data flow style, using scalar types for the definitions of state values, and delays for their memorization. If the modes are sequenced in a complex way however, then using an imperative style may be profitable.

Hence the association of an activity to a time interval is a basic element on the way to the definition of declarative, equational operators, in order to facilitate the expression of dynamical behaviors. These operators would be compatible with the equational approach of *SIGNAL*, and a formal calculus would be based on them, extending the model of *SIGNAL*.

An application currently being worked upon is the discrete sequencing of continuous, data flow tasks in active robotic vision [24].

In the following, in section 2, we introduce the notion of tasks, associating a data flow process and a time interval. We introduce the synchronous data flow language *SIGNAL* in section 3.1, and time intervals are introduced as an example of a particular *SIGNAL* process in section 3.2. Then, in section 4, the sequencing of time intervals is presented in the form of place-transition graphs, and various sequencing patterns. The sequencing of data flow tasks is handled in section 5. Section 6 exposes the basic ideas for an application of these principles to the sequencing of control functions in robotics. After a review of related work in section 7, we give perspectives of this work, concerning the programming language level aspects, as well as model-oriented and applications-oriented ones.

## 2 Tasks: associating a process with a time interval

### 2.1 Applications and processes

A data flow application is an activity that is executed over time i.e., non-instantaneously. It is composed of a set of data flow processes in the general sense: they may include memorizing and state information (e.g., such a process can be an automaton).

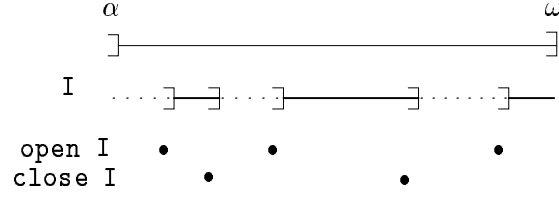
It is executed from an initial state that is built at an initial instant  $\alpha$ . This instant is not a reaction instant of the reactive process, in the sense that it is not a reaction to an input event: it is before the beginning of the reactive execution mechanism. Causally, this means that a process can not decide on its own starting (this point of view is also adopted in other languages e.g., ARGOS [14], cf. section 7).

A data flow process has no termination specified in itself. Therefore, its end can only be decided in reaction to the reception of external events or the reaching of given values. Such a process can be seen as an application that can be interrupted or aborted from outside: the termination is a reaction to an event, that is the last one treated by the application (e.g., the instant when the “QUIT” button is clicked in an application). This final instant can be called  $\omega$ . Causally, this means that the process can decide on its own termination (e.g., when reaching some threshold value), and also that it can perform terminal calculations within this last instant.

In this sense, a general data flow application executes on a time interval lasting from a beginning instant  $\alpha$  (not included as a reaction event) up to a final instant  $\omega$  (included in the interval). In other words, it executes on a left-open, right-closed interval  $] \alpha, \omega ]$ .

### 2.2 Time intervals

Our motivation in introducing time intervals is to enable the structured decomposition of the interval  $] \alpha, \omega ]$  into sub-intervals as illustrated in fig. 2. In this section, we define intervals by their basic properties given in several precise points below. These points will have to be complied with when encoding intervals in SIGNAL in section 3.2. In the context of data flow applications,

Figure 2: Decomposing  $] \alpha, \omega]$  into sub-intervals.

we are dealing with data in the form of series of values indexed by time: we call them signals e.g., signal  $X$  has values  $X_t$  (where  $t \in \mathbb{IN} \setminus \{0\}$  is not a date but a order index). In general, different signals can have values at different instants, especially events: they are not necessarily all present together. Intervals are defined in the following so that they have a value at any instant  $t$ .

An interval  $I$  is alternately inside (the solid lines in fig. 5) and outside (the dotted lines in fig. 5). Occurrences of the interval are the time intervals where the interval is inside. This state of the interval is expressed at each instant  $t$  between  $\alpha$  and  $\omega$  by its value  $I_t$ :

$$I_t \in \{\text{inside}, \text{outside}\}. \quad (1)$$

It is given an initial value from instant  $\alpha$  (in the example of fig. 2,  $I$  is initially **outside**). The complement of an interval  $I$  is the interval **compl**  $I$  which is **inside** when  $I$  is **outside** and reciprocally.

Coherently with  $] \alpha, \omega]$ , sub-intervals are taken to be left-open, right-closed; like in reactive automata, a transition is made according to a received event occurrence and a current state, which results in a new state: hence the instant where the event occurs belongs to the time interval of the current state, not to that of the new state. In other words, the value of  $I$  at time  $t$  is the value  $val_{t-1}$  which is determined by the occurrence of a bounding event at time  $t - 1$ :

$$I_t = val_{t-1} \quad (2)$$

The value  $val_t$  of the interval is given by a new value  $newval_t$  when there is one (i.e., when it is defined by an occurrence of a bounding event), or else

by the previous value:

$$val_t = \begin{cases} newval_t & \text{iff present} \\ val_{t-1} & \text{otherwise} \end{cases} \quad (3)$$

At the first instant  $t = 1$ ,  $val_0$  is defined by the initial value of  $I$ .

Finally, the new value  $newval_t$  alternates at the occurrences of opening event **open**  $I$  and closing event **close**  $I$ ; it is **inside** after the occurrence of **open**  $I_t$ , and **outside** after **close**  $I_t$ :

$$newval_t = \begin{cases} \text{inside} & \text{iff } (\text{open } I)_t \text{ is present} \\ \text{outside} & \text{iff } (\text{close } I)_t \text{ is present} \end{cases} \quad (4)$$

Furthermore, the point in defining sub-intervals is to express the restriction to a part of  $] \alpha, \omega ]$  of the existence of events and the data they carry with them: we want to distinguish occurrences of events that are inside and outside the interval:

$$\begin{cases} (X \text{ in } I)_t = X_t & \text{iff } I_t = \text{inside} \\ (X \text{ out } I)_t = X_t & \text{iff } I_t = \text{outside} \end{cases} \quad (5)$$

For an interval  $I := ]B, E]$  the opening bounds are outside of it: **open**  $I = B \text{ out } I$ , and the end bounds inside: **close**  $I = E \text{ in } I$ .

## 2.3 Tasks

The notion of *task* that we want to introduce consists in associating some (sub-)processes of the application with some (sub-)intervals (of  $] \alpha, \omega ]$ ) on which they are executed. Tasks active on interval  $] \alpha, \omega ]$  represent the default case: they are *remanent* throughout the whole application.

Inside the task interval, the task process is active i.e., present, and executing normally. Outside the interval, the process is inexistent i.e., absent, and the values it keeps in its internal state are unavailable. In some sense it is out of time, its clock being cut. Series of values of the signals involved in a task will be restricted to the time interval contained between the start and end event occurrences as seen in point (5) above.

This is transparent from the point of view of the process in the sense that if its internal state is kept, then re-entering a new occurrence of the task interval will see it resume execution as if it had remained active.

This introduces the characteristics defining a task:

- the process executed  $P$ ,
- the execution interval  $I$ ,
- the starting state (current, or initial) when (re-)entering the interval.

More precisely, this latter means that, when re-entering the task interval,

*the process can be re-started at its current state* (6)

i.e., with its internal memories set to their values at the instant where the task was *suspended* (meaning: in a temporary fashion). For a process  $P$  and an interval  $I$ , such a task can be noted  $\boxed{P \text{ on } I}$ .

Another way of specifying a tasks is that

*the process can be re-started at its initial state* (7)

as defined by the declarations of all its state variables, if the task was *interrupted* (meaning: aborted in a definitive fashion). This re-starting point of the process is given at the instant beginning the interval, which, like  $\alpha$  for applications, is outside of the execution interval. For a process  $P$  and an interval  $I$ , such a task can be noted  $\boxed{P \text{ each } I}$ .

A suspension or interruption occurs at the end instant, which is inside the interval, like  $\omega$  in the case of applications. Hence, on each end instant, particular treatments can be performed (signals emitted, memories updated, ...). It also means that the end of the execution interval can be determined from inside the process, in a locally data-driven way, as well as from outside of the task.

The process in a task can itself feature sub-tasks, which are active on sub-intervals of the task interval. This introduces the possibility of hierarchical arrangements of tasks. Parallelism between tasks is obtained naturally when tasks share the same interval, or overlapping intervals. Other structures for the sequencing of tasks are studied later in this paper.

For example, consider a process `count` with input signal  $T$ , and output signal the number  $N$  of occurrences of  $T$ . We want to specify a task counting a certain number  $V$  of occurrences of  $T$  after event  $B$ . For this, the process `count` can be associated with the interval beginning with event  $B$  and ending with an event  $E$  defined by the moment when  $N$  takes the value  $V$  as follows:

`count each ]B, E]`

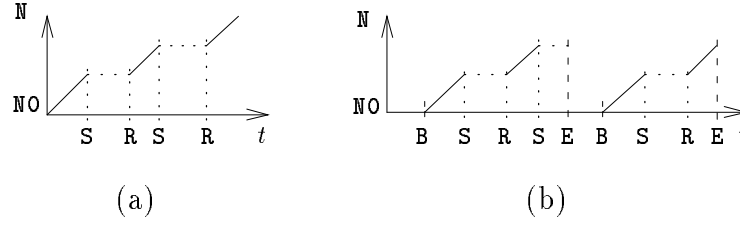


Figure 3: Counter suspended on  $]S,R]$  (a), each  $]B,E]$  (b).

which provides us with an example of a process determining its own termination: the calculated output value participates in the decision of ending the execution interval.

## 2.4 Derived task behaviors

Task behaviors different from the two constructions **on** and **each** introduced earlier, and less primitive, are considered in this section.

### 2.4.1 Suspending and terminating a task

For example, consider a counter of the number  $N$  (initialized to the value  $NO$ ) of occurrences of an input, and is:

- *started* upon begin event  $B$ , with initial value  $NO$ , and
- *stopped* upon end event  $E$ ,
- *suspended* upon event  $S$ , until it is
- *resumed* upon event  $R$ .

The suspension of the process **count** on interval  $]S,R]$  means that it is active on the complementary interval (using a complementation operator **compl**, we call it **compl ]S,R]**). This is obtained by:

$$\text{count on compl } ]S,R]$$

which behaves as illustrated in fig. 3 (a). Counting is performed on the complement of  $]S,R]$  (relatively to  $]\alpha,\omega]$ : those occurrences of the input that are *outside* of  $]S,R]$  are counted i.e., *inside* the interval **compl ]S,R]**.

Its starting and ending on interval  $]B,E]$  is then obtained by:

`(count on compl ]S,R]) each ]B,E]`

which behaves as illustrated in fig. 3 (b). The counter is re-initialized to the value `N0` each time the interval  $]B,E]$  is re-entered. In particular, counting can be stopped whether it is suspended or not.

#### 2.4.2 Suspending activity, with values available

An activity (e.g., setting alarm time in an alarm clock, ...) can be suspended, while the computed values should remain accessible to other components of the application (e.g., for display, comparison of alarm time with current time, ...) Then, it is the computing activity that must be suspended, while the memorizing of values remains active: they must be separate tasks, each associated with its interval.

The following example concerns a stopwatch, and especially the alarm setting functionality: the alarm time is set (i.e., incremented) by pushing a button during interval `Alarm_update` delimited by occurrences of events from another button of the watch. This alarm time is displayed on the watch display only during an interval `Show_Alarm`, but is constantly compared to the current time, given by the timer, and outputting an event `A` causing the bell to ring in case of equality.

Thus, the alarm time, memorized in a variable `M`, fed by the value `V`, is modified by a process `update` on interval `Alarm_update`, is displayed by a process `display` on interval `Show_Alarm`, and compared to the current time `T` by a remanent process `compare`. An encoding for this can be sketched as follows:

```
(|  V  := update{M} on Alarm_update
|  D  := display{M} on Show_Alarm
|  A  := compare{M,T} % remanent %
|  M  := memorize{V}  % activities %
|)
```

#### 2.4.3 Restarting anew on each beginning event

Another particular behavior involves stopping a task and restarting it in its initial state i.e., a new instance of it, *within the same instant*, as ESTEREL and



ARGOS do (cf. section 7). An example of application of this is the stopwatch, where the reset can be seen as the stopping of the stopwatch, and within the same instant, the start of a new instance of it.

This involves performing two kinds of actions upon the occurrence of this **Reset** event: computing the values at this instant inside the interval, and preparing the next interval occurrence i.e., re-initializing state variables. There are different approaches to integrate this behavior into our framework:

- it is possible to define a **restart** primitive, along with **on** and **each**.

However, this involves a modification of the definition of intervals, because they should have occurrences meeting at bounding instants, which is not the case by now (where there is at least one instant between two interval occurrences); the interpretation of the intervals as tasks suggests that it might be envisaged. But then, it divides an interval into parts  $]B, B]$  followed by a final part  $]B, E]$ , and the question arises whether they must all be considered as separate interval occurrences of  $I := ]B, E]$ , or just as parts of one occurrence from  $B$  to  $E$ .

- it is possible to simulate it using two intervals, as follows:

```
(|  IB := ]B,B]
|  P each IB
|  P each compl IB % complement relative to ]B,E] %
|)  each ]B,E]
```

but this approach involves, in a sense, duplicating the process  $P$ .

- resetting the internal state of processes can be considered to be independent of the concept of interval, and achieved using a process input for the reset event, managed manually by the programmer. The reset then happens to coincide with the beginning event of the interval. Hierarchically, it can involve propagating the reset to sub-processes, going all the way down to each memory cell or delay.

However, this solution provides no general structure.

The possibilities in this alternative require closer examination before choosing whether having a derived features of the language, or a primitive.

### 3 SIGNAL, time intervals and tasks

In this section we show how the concepts presented above can be specified and integrated in the data flow synchronized language SIGNAL. We will first introduce the essential mechanisms of the SIGNAL language, and then the time intervals, which will serve as an example of SIGNAL process, encoding the memorizing of state information [18]. On these bases, possible encodings of the tasks in SIGNAL are proposed.

#### 3.1 A brief introduction to SIGNAL

##### 3.1.1 An equational synchronized data flow language

SIGNAL [18] is an equational, data flow oriented, synchronous language, built around a minimal kernel of basic constructs. It manipulates signals, which are unbounded series of typed values, with an associated clock determining the instants where values are present; for instance, a signal  $X$  denotes the sequence  $(x_t)_{t \in T}$  of data indexed by time  $t$  in a time domain  $T$ . Signals of a special kind called **event** are characterized only by their clock i.e., their presence (they are given the boolean value **true** at each occurrence); given a signal  $X$ , its clock is obtained by the expression **event**  $X$ , giving the event present simultaneously with  $X$ . The constructs of the language can be used to specify, in an equational style, relations between signals i.e., between their values and between their clocks. Systems of equations on signals are built using the composition construct.

The compiler performs the analysis of the consistency of the system of equations, and determines whether the synchronization constraints between the signals are verified or not. If this is the case, and if the program is constrained enough so as to compute a deterministic solution, then executable code is produced.

From a data flow point of view, SIGNAL processes communicate through signals, seen as sequences of typed and timed values. A small number of elementary processes, and the possibility to link them into a network, are used to build larger applications, using the hierarchical process structure [8]. Within this framework, the SIGNAL programming environment features a graphical, block-diagram oriented editor, allowing for top-down as well as bottom-up design [7].

### 3.1.2 The kernel of SIGNAL

Basic operators in SIGNAL define elementary processes, each of which corresponds to an equation; they are:

**functions:** they are defined on the types of the language (e.g., boolean negation of a signal  $E$ : `not E`). The signal  $(Y_t)$ , defined by the instantaneous function  $f$  in:  $Y_t = f(X_{1t}, X_{2t}, \dots, X_{nt})$  is written:

$$\boxed{Y := f\{ X1, X2, \dots, Xn \}}$$

Functional expressions are monochronous, meaning that the signals  $Y$ ,  $X1$ , ...,  $Xn$  are said to be synchronous: they share the same clock. In other terms, when computing the value of  $Y_t$ , all  $X_i$  are taken with their value at that time  $t$ ; therefore they are all required to have the same clock.

**delay:** it gives the past value of a signal, generally noted  $ZX_t = X_{t-d}$ , with initial values  $ZX_i = V_i$ , for  $0 \leq i < d$ ; in SIGNAL, for the simple case where  $d = 1$ , this is written:

$$\boxed{ZX := X\$1} \text{ with initialization } \boxed{ZX \text{ init } V0}$$

Delay is monochronous too i.e.,  $X$  and  $ZX$  have the same clock.

**conditional selection:** under-sampling a signal  $X$  according to a boolean condition  $C$  is written in SIGNAL:

$$\boxed{Y := X \text{ when } C}$$

This operator is polychronous: the operands and the result do not have identical clock. Signal  $Y$  is present if and only if  $X$  and  $C$  are present at the same time and  $C$  has the value `true`. Thus,  $Y$  is at most as frequent as  $X$  and  $C$ : in other terms it is non-strictly less frequent than both of them. The intersection of the clocks of  $X$  and  $C$  (i.e., the instants when the expression can be evaluated) includes the clock of  $Y$  (which features only the instants when  $C$  evaluates to `true`). When  $Y$  is present, its value is that of  $X$ .

**deterministic merge:** defining the union of two signals of the same type is written:

$$\boxed{Z := X \text{ default } Y}$$

This operator is polychronous too: the clock of  $Z$  is the union of that of  $X$  and that of  $Y$ : thus it is at least as frequent as each of them.

The value of  $Z$  is the value of  $X$  when it is present, or else that of  $Y$  if it is present and  $X$  is not.

**process composition:** elementary processes can be composed with the associative and commutative operator “ $|$ ” denoting the union of the underlying systems of equations. In SIGNAL, for processes  $P_1$  and  $P_2$ , it is written:

$$\boxed{(| P_1 | P_2 |)}$$

For example, the equation  $x_t = x_{t-1} + 1$  can be written:

$$\begin{cases} x_t &= z x_t + 1 \\ z x_t &= x_{t-1} \end{cases}$$

which is written in SIGNAL:  $(| X := ZX + 1 | ZX := X\$1 |)$ .

Furthermore, it is possible to confine signals locally to a process using “/”: e.g., in the previous example, hiding  $ZX$  gives the following code:  $(| X := ZX + 1 | ZX := X\$1 |)/ZX$ .

### 3.1.3 Modularity and derived processes

The language is built upon this kernel, and features derived operators for a variety of facilities, like arrays or variables. A structuring mechanism is proposed in the form of the definition of process schemes, which are given a name, typed parameters, input and output signals, a process, and local declarations. Occurrences of process schemes in a program are expanded by the compiler. Derived processes have been defined from the primitive operators, providing programming comfort [8], such as: **synchro** $\{X, Y\}$  which specifies the synchronization of signals  $X$  and  $Y$ ; **CLK**  $:=$  **event**  $X$  giving the clock  $CLK$  of a signal  $X$ ; **when**  $C$  giving the clock of occurrences of  $C$  at value **true**; and **X cell**  $B$  which memorizes values of  $X$  and outputs them also when  $B$  is true. Arrays of signals and of processes have been introduced as well.

### 3.1.4 Compilation and clock calculus

We only give a very brief overview of the compilation process, the formal calculus underlying it, as well as the formal analysis of static and dynamical properties of systems. Some of the perspectives exposed in section 8 will refer to these aspects of SIGNAL.

In order to model the absence or presence, and the value **true** or **false**, of logical signals, an encoding in  $\mathbb{Z}/3\mathbb{Z}$  has been defined, as well as formal calculi on this model [18]. It is as follows: absence is coded by 0, presence with value **true** is coded by +1, presence with value **false** is coded by -1. For a non logical signal, its value is not encoded: only its presence or absence is (by  $\pm 1$ ). Hence, for a signal  $X$  with value  $x$ , its clock is encoded by  $x^2$  (as it is equal to 1 when the signal is present).

Clock equations for primitive operators are encoded as follows:

- $X := f\{X_1, \dots, X_n\}$  by:  $x^2 = x_1^2 = \dots x_n^2$  (clocks are equal)
- $Y := X\$1 \text{ init } X_0$  by:  $y^2 = x^2$  (clocks are equal)
- $Y := X \text{ when } C$  by:  $y^2 = x^2(-c - c^2)$  (i.e.,  $y^2 = x^2$  iff  $c = 1$  and  $c^2 = 1$ )
- $Z := X \text{ default } Y$  by:  $z^2 = x^2 + (1 - x^2)y^2$  (i.e.,  $z^2 = x^2$  iff  $x^2 = 1$ , or else  $z^2 = y^2$  iff  $x^2 = 0$ )

For a signal defined by a delay, its past value is taken into account when calculating the present one. This involves a dynamical system; for the process  $X := V\$1 \text{ init } V_0$ , with  $V$  boolean:

$$\begin{cases} \xi' = v + (1 - v^2)\xi \\ \xi_0 = v_0 \\ x = v^2\xi \end{cases}$$

where  $\xi$  represents the current state of the system,  $\xi'$  the next state, and  $\xi_0$  its initial value. The value of the current state changes when  $v$  is present, otherwise (i.e.,  $v = 0, (1 - v^2) = 1$ ) it keeps its value  $\xi$  at instants of a clock more frequent than  $v^2$ .

Using this encoding, a system of equations can be associated with any SIGNAL program, and the corresponding temporal behavior can be analysed, and possible inconsistencies detected. This is called the *clock calculus*: it has a static part, and a dynamical part. The static part is performed on a graph

of data dependencies between the signals of the program, each conditioned with the clock at which it is effective. This graph enables the detection of dependency cycles, where the conditioning clocks are not exclusive, meaning that there exist instants where the dependency cycle is effective. The graph is also used for other treatments on the programs: code generation, optimization, partitioning and placement on multi-processor architectures [18, 19]. The analysis of the dynamical behavior of the programs relies on the equational representation of the synchronization and of the logic of programs in  $\mathbb{Z}/3\mathbb{Z}$ : it consists in translating properties of the program into equivalent algebraic properties, that can be verified by formal calculus tools [11]; this is implemented in the proof tool SIGALI. Other works concern the automatic synthesis of hardware architectures implementing SIGNAL programs [35]. In relation with other languages of the synchronous approach (LUSTRE and ESTEREL), common formats have been defined, in the perspective of sharing the various tools that grew around them [27].

### 3.2 Time intervals in SIGNAL

The time interval is an example of a SIGNAL process [18, 4]. The encoding of time intervals in SIGNAL is made by considering each of the points in section 2.2.

The values of intervals defined in point (1) are encoded as a boolean signal: **inside** is **true** and **outside** is **false**. The initial value is given using the SIGNAL statement **init** in the declaration of the state variable encoding the interval. The complement **compl I** of an interval **I** is the boolean negation **not** of **I**.

Intervals are left-open and right-closed<sup>2</sup>, as stated in point (2): this is encoded as: **I := Val \$ 1**.

Following point (3) the value **Val** is the new value **New\_val** when it is present, or its previous value (i.e., **I**) otherwise, which is encoded as follows: **Val := New\_Val default I**.<sup>3</sup>

---

<sup>2</sup>Other forms of intervals, **[B,E[**, **[B,E]**, and **]B,E[** can also be envisaged, but we chose **]B,E]** as a standard for reasons given in section 2.1.

<sup>3</sup>This is essentially equivalent to the macro-construct of SIGNAL called **V cell C** [8], which is a memory cell outputting the latest value of **V** when the condition **C** is present and true. The difference is that it outputs the new value of **V** when it is present; in the interval, it is the previous value that is output.

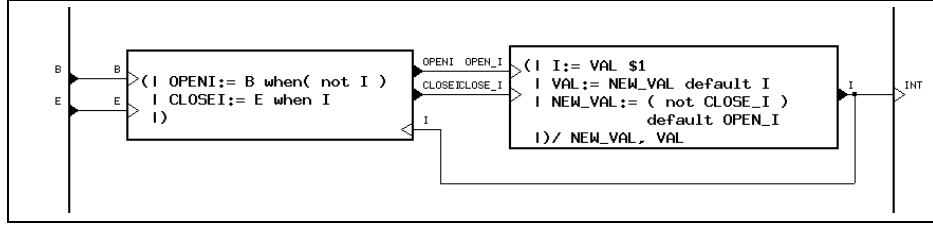


Figure 4: The time interval programmed with the graphical interface.

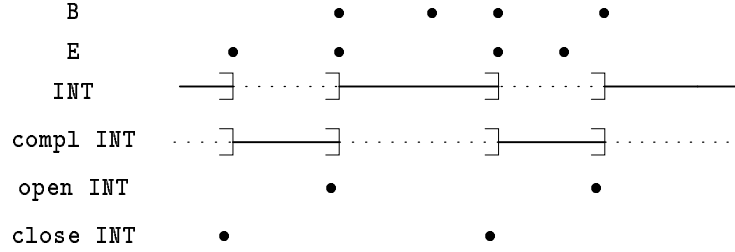


Figure 5: A possible trace of a time interval and its bounding events.

The new value alternates at the occurrences of bounding events as defined in point (4): `New_Val := (not (close I)) default (open I)`.

The restrictions for a signal `X`: `X in I` and `X out I` of point (5) are respectively encoded as: `X when I` and `X when not I` (i.e., `X in compl I`).

To summarize, a left-closed, right-opened interval `INT`, with opening event `B` and closing event `E` noted:

$$\boxed{INT := ]B, E]}$$

is encoded as shown in fig. 4, obtained with the graphical interface of the SIGNAL environment [7], and where boxes correspond to levels of composition. The bounds `open I` and `close I` are defined respectively by `B out I` and `E in I`. The trace shown in the chronogram in fig. 5 for an interval initially *inside* illustrates in particular that when `B` and `E` occur simultaneously, priority is given to changing the state. Also, when *outside*, receiving `E` has no effect, and reciprocally when *inside* and receiving `B`. It behaves like the two-state automaton illustrated in fig. 6. The interval process recognizes series of occurrences of the interval defined by occurrences of its bounding events `B` (begin) and `E` (end).

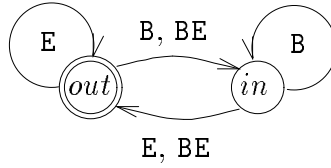


Figure 6: The time interval as a two-state automaton.

Coherently with the boolean encoding of *inside* and *outside*, the union of intervals `I1 union I2` can be coded as the boolean disjunction `I1 or I2`, intersection `I1 inter I2` as the conjunction `I1 and I2`. This brings forward the question of the clocks of intervals, in particular with regard to the constraints imposed by functional operations in SIGNAL (see section 3.1). As a SIGNAL process, the interval has no fully defined clock: the fact that it is defined `default` its previous value makes that the only constraint is that it be at least as frequent as `(not (close I)) default (open I)`. The clocks are being resolved by the compiler i.e., the clocks at which the values of the intervals must be computed is the upper bound of all the clocks at which it is needed i.e., the clocks of the expressions in which it is involved. The calculation of this upper-bound is a special treatment which distinguishes intervals from regular signals: it argues in favour of the constitution of a specific type for intervals.

### 3.3 The encoding of tasks in SIGNAL

The encoding of tasks in SIGNAL can be considered at different levels, from the lowest and most general, to a higher-level and less costly one. It has to comply with points (6) and (7) of section 2.3.

#### 3.3.1 Primitive processes level

At the level of the five instructions of the SIGNAL kernel, the encoding offers generality, being applicable to any SIGNAL process. An advantage is that this is precisely the level at which the re-initialization of state variables (or delays) has to be performed. Also, this is the level where the compiler manages all the dependencies, optimizations, etc ... A possible drawback is that the



P	P on I	P each I
$Y := f\{X_i\}$	$Y := f\{X_i \text{ in } I\}$	
$Y := X \text{ when } B$	$Y := (X \text{ in } I) \text{ when } (B \text{ in } I)$	
$Y := X1 \text{ default } X2$	$Y := (X1 \text{ in } I) \text{ default } (X2 \text{ in } I)$	
$(  P1   P2  )$	$(  P1 \text{ on } I$ $  P2 \text{ on } I  )$	$(  P1 \text{ each } I$ $  P2 \text{ each } I  )$
$Y := X\$1 \text{ init } V0$	$Y := (X \text{ in } I)\$1$ $\text{init } V0$	$(  Y\_V :=$ $(V0 \text{ when open } I)$ $\text{default } (X \text{ in } I)$ $  Y := (Y\_V\$1 \text{ init } V0)$ $\text{when event } (X \text{ in } I)$ $ )$

Table 1: Primitives-level encodings of **on** and **each**.

encoding is performed on the expanded form of the process instances, and can introduce redundancy in signals filtering.

**Encoding.** Encodings for **on** and **each** are proposed in table 1. Simplifications of this encoding are possible e.g.,  $(X \text{ in } I) \text{ default } (Y \text{ in } I)$  is equivalent to  $(X \text{ default } Y) \text{ in } I$ , in terms of signals, even if the executable code produced by the SIGNAL compiler makes a difference between filtering before or after computing the value of an expression.

**Example.** Intervals as defined in section 3.2 are particular cases of processes:  $I := ]B, E]$  has inputs B and E, and logical output I. Hence, an interval can be associated with another interval into a task. In table 2 we give the encodings of  $I \text{ on } J$  and  $I \text{ each } J$ . The code proposed in section 3.2 for interval  $I := ]B, E]$  is arranged in order to dissociate the computation of the new value of the interval NI, from the memorization of its previous value in ZI

The parts of the code that are added to this specification in order to transform it into  $I \text{ on } J$  and  $I \text{ each } J$  are framed in boxes.

Note that the in J filtering in the NI line could be factorized as it is in the VI line, which is one of the simplifications applicable to the code obtained from applying schemes in table 1.

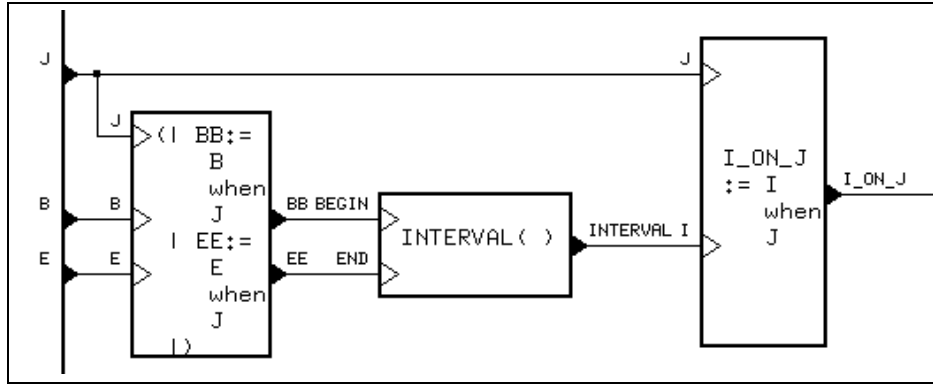


Figure 7: Filtering input and output of P on I: example of an interval.

### 3.3.2 Task process level

**Encoding on.** At that level, the encoding of *on* consists in filtering the inputs and outputs of the process in the interval. The advantage of this highest level is that it avoids duplicating filterings down to the lowest levels, as illustrated in fig. 7.

For example, the interval  $I\_ON\_J := ]B, E]$  on J is encoded by filtering input events (i.e., bounding events) inside interval J, and by filtering the output (i.e., the value of the interval) as follows:

$$I\_ON\_J := ]B \text{ in } J, E \text{ in } J] \text{ in } J.$$

This is equivalent to the coding proposed in table 2: only, the filtering is made at the interface of the process with the external world, as fig. 7 shows, instead of the primitive processes level.

Filtering both inputs and outputs is motivated by the existence of processes whose clock is determined by constraints imposed by their context. This can imply the possibility of state or value changes in the absence of inputs. Hence, in order to avoid that, when the clock is defined from that of the outputs, those must be filtered to be present only *inside* the interval.

**Encoding each.** Concerning the encoding of *each* at this level, the reinitialization needs to be propagated at lower levels: all state variables must be provided with a reset signal, defined in terms of all the reset signals along the hierarchical structure.

I1 := ]B,E] on J	I2 := ]B,E] each J
<pre> (  NI :=((not E) in I) in J   default B in J   VI :=(NI default ZVI) in J   ZVI := (VI in J)\$1 init IO    I1 := ZVI  ) </pre>	<pre> (  NI :=((not E) in I) in J   default B in J   VI :=(NI default ZVI) in J   ZVI_V := ((IO when open J)   default VI in J)   ZVI := (ZVI_V\$1 init IO)   when event (VI in J)   I2 := ZVI  ) </pre>

Table 2: Encodings of I on J and I each J.

For example in the process (`| X := expr | ZX := X$1 |`), X has the value of expression *expr*, and ZX has the value at the previous instant of X. When re-initializing the state variable, the value to be input in the delay is not the current value of X (i.e., of *expr*), but its initial value X0. That is to say, the value of ZX must be the previous value of X, except when **Reset** is present: then it must be X0. In SIGNAL, this is written:

`ZX_V := (X0 when Reset) default X.`

This signal has a clock more frequent than that of X in its original definition. Therefore this value, after having been delayed, must also be filtered in order to exactly have the clock of X i.e., **event** X.

Hence, a delay process with a **Reset** can be coded as follows:

```

(|   X   := expr
|   ZX_V := (X0 when Reset) default X
|   ZX   := (ZX_V$1) when (event X)
|)

```

where `ZX init X0`.

This resetting scheme can be applied to the encoding of **each**, with **Reset** being **open** I i.e., **B out** I. It can also serve for the encoding of the restarting on each occurrence of the beginning event presented in section 2.4.3, for which **Reset** is B.

## 4 Sequencing time intervals

Now that tasks are defined by the association of a process with an interval, we want to proceed towards sequencing them. An intermediary stage is to consider the sequencing of time intervals. Each time interval holds some state information, and events cause transitions between these states. Hence one way to specify a sequencing is to give the corresponding automaton, or a place-transition system where a transition can lead to several places simultaneously.

The SIGNAL language is event-based, hence relations between intervals will be constructed on their bounding events. These events fire the transitions between states or places; there exist methods to constrain them in SIGNAL, so that the resulting intervals verify some constraints, such as being disjoint, or in strict sequence.

We first describe how a transition between two places can be specified, and then how parallelism can occur between places, hence enabling the specification of place-transition systems. As particular cases of this, we will then describe various patterns for the sequencing of two intervals.

### 4.1 Place-transition networks

Automata and place-transition systems are widely spread formalisms for the specification of dynamical behaviors. As models, they underlie many other, higher-level formalism, including the synchronous languages like ESTEREL. Also SIGNAL supports the specification of automata. What we present here is one way of programming them using time intervals. However, it neither means that this is the best or only way to program automata in SIGNAL, nor that dynamical behaviors should be programmed in terms of automata, as discussed in section 4.2.

In the simplest case, a transition from state **S1** to state **S2** on the occurrence of an event **E**, as illustrated in fig. 8, can be coded as follows:

$$S1 := ] A , E ] \mid S2 := ] E \text{ in } S1 , B ]$$

Associating an instantaneous action to a transition, as is usually done in automata, can be done here by conditioning the action by the presence of the transition event. For example, emitting event **0** on the transition can be

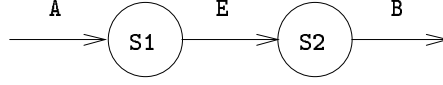


Figure 8: A transition between two states.

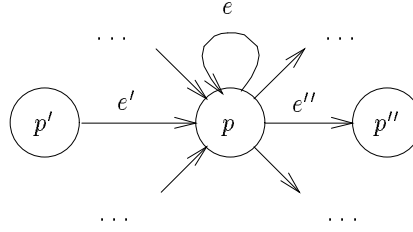


Figure 9: A place with incoming and outgoing transitions.

done in any the three following ways:

`0 := when E in S1, or when close S1, or when open S2.`

In an automaton, the control state is located in one only place. In a place-transition network, it can be in several places at ounce, which means that the control state is defined by the set of entered places. There can be several simultaneous transitions if several transition events are present simultaneously, or if a same event causes several transitions. In the general case, there can be several incoming or outgoing transitions to a place, labelled by different events, as illustrated in fig. 9.

A place-transition network  $\langle P, L, T \rangle$  is defined by a set  $P$  of places  $p$ , a set  $L$  of labels  $e$  (in our case: events) and a set  $T: T \subseteq P \times L \times P$  of transitions  $t$  of the form  $t = \langle p', e, p'' \rangle$ . For each transition  $p \in P$  we have an interval  $I_p$  defined by  $] \text{Begin}_p, \text{End}_p ]$ . The beginning event  $\text{Begin}_p$  is defined as the union for all  $\langle p', e', p \rangle \in T$  of occurrences of events  $e'$  when in place  $p'$  i.e., the `default` of  $(e' \text{ in } I_{p'})$ :

$\text{Begin}_p := \text{default}_{\langle p', e', p \rangle \in T} (e' \text{ in } I_{p'}).$

The end event  $\text{End}_p$  is defined in terms of the outgoing transitions i.e., the union for all  $\langle p, e'', p'' \rangle \in T$  of occurrences of events  $e''$ :

$$\text{Outgoing}_p := \text{default}_{\langle p, e'', p'' \rangle \in T} e''.$$

However, in case some incoming event is present simultaneously, the place should not be exited: therefore, the end event is more restrictively defined, excluding this case<sup>4</sup>:

$$\text{End}_p := \text{Outgoing}_p \text{ without } \text{Begin}_p$$

To summarize, each place  $p$  is coded as follows:

```
(| Ip := ] Beginp, Endp ]
| Beginp := default\langle p', e', p \rangle \in T (e' in Ip')
| Endp := (default\langle p, e'', p'' \rangle \in T e'') without Beginp
|)
```

The initial state of the place-transition network is given by initializing the corresponding interval *inside* (i.e., at the value **true**), and others *outside*.

A remark can be made regarding transitions returning on the same place, like the one labeled by  $e$  in fig. 9. Such transitions encountered in automata are used to mean that a certain action is performed on this transition, without changing the state. In our framework, actions are separated from the state-transition system, as we will see further when presenting the task structure; hence this kind of transition, leaving the state unchanged, is useless, and should better be avoided in automata (as illustrated in the example in section 4.2). However, in place-transition systems, this kind of transitions can be interpreted as restoring a token in the place  $p$  while entering place  $p''$ . In our case, this means that  $I_p$  should not be exited when  $(e \text{ in } I_p)$  is present: this is taken care of by the use of **without** in the definition of  $\text{End}_p$ .

We can illustrate these bases by using them for the specification of various ways of sequencing intervals. This is done by filtering bounding events, so that resulting intervals comply the constraints by construction. We will define flat structures, in the sense that they have no hierarchical structure: hierarchies are introduced further, for the sequencing of tasks.

---

<sup>4</sup>We define **C without A** by the expression: **C when ((not A) default C)**.

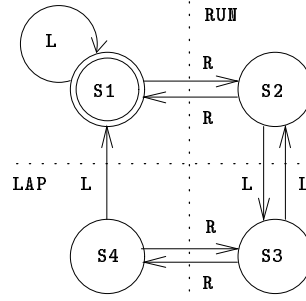


Figure 10: Automaton for the logic control of a stopwatch.

## 4.2 Example: the control logic of a stopwatch

**Automaton specification.** The example of the control logic of a stopwatch is well known in the synchronous languages literature [14, 15]. It counts impulsions of a base clock, and is controlled by two buttons *R* (for run) and *L* (for lap), emitting corresponding events. For simplicity purposes, these events are supposed to be exclusive i.e., they are never simultaneous. Pressing button *R* causes alternately running and stopping the stopwatch. Pressing *L* freezes or un-freezes the display of current time, except when the stopwatch is stopped and the display un-frozen: in that case, it is interpreted as a reset button. This behavior is described by the automaton illustrated in fig. 10. We can apply what was described above, simplifying the scheme for this automaton: it can be coded in *SIGNAL* intervals as follows:

```
(|  S1      := ]R in S2 default L in S4, R]
|  S2      := ]R in S1 default L in S3, R default L]
|  S3      := ]R in S4 default L in S2, R default L]
|  S4      := ]R in S3, R default L]
|  Reset   := L in S1
|)
```

where intervals are initialized *outside* (i.e., at the value **false**), except *S1*, which is the initial state, initialized *inside* (i.e., **true**).

In particular, the reflexive transition on *S1* labeled by *L* is not encoded as a transition here, but as an action (emitting **Reset**) performed on each occurrence of *L* in *S1*.

**Equational specification.** However, this coding of the automaton is unsatisfying, because it does not benefit from the advantages of SIGNAL programming in the design of this behavior: it is too close to the automaton, and therefore lacks hierarchical structure. Another method, closer to the equational approach specific to SIGNAL, is to specify the aspects of the behavior in a declarative way, and let the SIGNAL compiler analysing the corresponding constraints, in order to build the implicit automaton. In this example, it consists in exploiting the symmetry, visible in fig. 10, in characterizing the stopwatch by the fact that:

- it counts between two occurrences of R,
- it freezes the display of time between two occurrences of L
- it resets the counter on L when not running nor freezing the display.

A corresponding encoding of this behavior is as follows:

```
(|  RUN      := ]R, R]
|  LAP      := ]L in RUN, L]
|  Reset    := (L out RUN) out LAP
|)
```

The associated tasks can be specified as follows:

```
(|  Time_value := (count on compl RUN) each ]R, Reset]
|  Time_display := memorize{Time_value} in compl LAP
|)
```

### 4.3 Particular sequencing patterns

Various classical sequencing patterns can be specified that way i.e., defined in the form of states and transitions between them on the occurrence of events; for instance:

- sequence is a simple transition, as seen above in section 4.1;
- a cobegin-coend parallelism is achieved by entering two places simultaneously:

```
(| I1 := ]B,E1] | I2 := ]B,E2] |)
```

i.e., two intervals overlapping from their begin instant;



- a loop is a cyclic path in the transition graph;
- a kind of watchdog, having I1 exited into I2 in case event W occurs before its “normal” termination by E:

(| I1 := ]B,W default E] | I2 := ]W in I1,C] |).

However, a less classical approach in the specification of sequencing patterns can be adopted e.g., by taking into account the periodical behavior<sup>5</sup> of the primitive interval, which can be considered as a form of endless loop. For example:

- the alternance of occurrences of intervals delimited by events A, B and C, D can be coded as follows:

```
(|  I1 := ](A out I) default (A when close I2), B]
  |  I2 := ](C in I) default (C when close I1), D]
  |  I := ] close I1, close I2]
  |)
```

where I is used as auxiliary interval: no new occurrence of I1 can begin *inside* I (and reciprocally no new I2 *outside* I).

- occurrences of intervals can be required to be disjoint; when both opening events A and C are present, priority is given to beginning the interval opened by event A (i.e., interval I1), so that this behavior is deterministic. For this purpose, we define C *without* A by the expression C **when** ((not A) **default** C). The coding is:

```
(| I1 := ](A out I2) default (A when close I2), B]
  | I2 := ]((C without A) out I1)
                                default (C when close I1), D]
  |)
```

This pattern is less constrained than the previous one, as it does not impose alternance.

These intervals may have common bounds, but, being semi-closed, they do not overlap.

---

<sup>5</sup>It is periodical in the sense that an interval denotes a series of occurrences; it is alternatively, and repetitively entered and exited again: hence it can be called periodical in a qualitative sense; a quantitatively scaled period is not what is meant here.

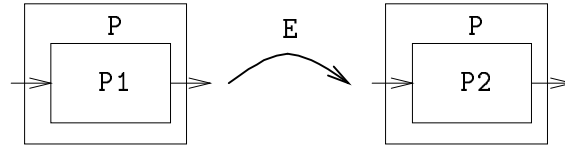


Figure 11: A task P behaves like P1 until event E, and then like P2.

## 5 Sequencing data flow tasks

A task is defined as being a process associated with an execution interval: sequencing such tasks means actually constraining their execution intervals.

One possibility is to use the sequencing patterns exposed above, and associate corresponding intervals to tasks. Another approach is to define composition operators on processes, closer to imperative composition operators (sequential, parallel composition, ...). As was said earlier, a data flow process has no specified termination in itself, and its end as well as the transition to another task, can only be decided in reaction to the occurrence of an event; hence the sequencing of data flow tasks has to be reactive.

### 5.1 Process composition

A quite natural, or at least very current sequencing pattern, is sequence itself, meaning that a process P can be defined as behaving as illustrated in fig. 11, first like process P1 until the occurrence of event E, and thereafter like process P2. This can be written:  $\boxed{\text{P1 until E then P2}}$ , and simply encoded as follows:

```
(| I := ]E, E'] init false
  | P1 on compl I
  | P2 on I
  |)
```

where the end E' of interval I can remain unspecified: the termination of P2 will be that of P, which is specified at a higher hierarchical level. The transition event E can be defined from values computed by process P1, or be an input of process P.

The definition of the interface of the new process P is made as for other operators [13]: the inputs of process P are the union of the inputs of P1

and those of P2; the output signals of P are the union of those of P1 and P2: for these latter, it can be the case that outputs of P1 and P2 have a non-empty intersection: the resulting output of P is simply the **default** of both, which will never be present simultaneously anyway, as **I** and **compl I** are necessarily disjoint. This operator resembles perhaps more the “chop” operator found in certain interval-based specification languages [29], than the classical sequential composition of ALGOL-like programming languages, because of its reactivity.

It can be used to define a sequential composition, for example in:

**P1 until E1 then (P2 until E2 then P3)**

where each of the three processes will be executed in the given order. In contrast, another combination of such structures behaves as a watchdog-like interruption structure; in:

**(P1 until E1 then P2) until E2 then P3**


if **E2** occurs before **E1**, process **P1** will be interrupted, and **P2** will be skipped, the application transiting directly to **P3**.

Other such processes composition operators can be defined e.g., corresponding to imperative control structures (parallelism, condition, ...).

## 5.2 Modularity and hierarchy

In general, it is possible in this framework to achieve the specification of a desired reactive, dynamical behavior with a separation of the definitions of:

- the internal behavior of the various processes,
- their individual association with execution intervals, into a task, using **on** or **each**,
- their sequencing (i.e., constraints on bounding events: external, or internal on sub-tasks output values).

The resulting process, as illustrated in fig. 12, has inputs and outputs that are the union of those of their components. If intervals are not disjoint, then several sub-tasks can simultaneously produce values for the outputs of P. In that case, a unique values for output signals must be determined by an operation marked by the  box in fig. 12. For example:

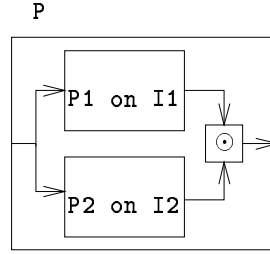


Figure 12: A process P with sub-tasks P1 and P2.

- a deterministic priority can be given between the processes (e.g., using `default`),
- there could be a composition function on the values of signals (e.g., for integers or reals: `+`, `*`, for booleans: `or`, ...), determining the value of P's output signal (as exists in ESTEREL [6]).

The sequenced processes can be sequencings of sub-tasks themselves: hierarchical, parallel automata can be programmed, allowing for various interruption structures.

For example, the stopwatch of which the automaton is presented in fig. 10, is part of a broader watch [14]. The complete watch, for which the control automaton of the upper level is illustrated in fig. 13 (with `ALARM_UPDATE` in expanded form), features the following modes:

- the `TIMER` mode, the initial one, where time is displayed, and
  - the `ll` button changes to `STOPWATCH` mode,
  - the `ul` button changes to `TIME_UPDATE` mode,
  - the `lr` button switches the time display mode between `24H` and `AM_PM`: to this end it emits event `switch_display_mode`,
  - the `ur` button switches the light on, emitting `switch_light_on`.
- the `TIME_UPDATE` mode, where:
  - the `ll` button changes the updated component of time (year, month, day of month, day of week, hour, minutes, tens of minutes, seconds), emitting `change_item`,

---

```

(| % switching off the alarm bell %
| stop_bell := ur
| % definition of the intervals %
| (| I_timer := ] ll in I_alarm default ul in I_t_update,
|                                     ll default ul ] init true
| I_t_update := ] ul in I_timer, ul]
| I_stopwatch := ] ll in I_timer, ll]
| I_alarm := ] ll in I_stopwatch default ul in I_a_update,
|                                     ll default ul ]
| I_a_update := ] ul in I_alarm, ul]
|)
| % tasks, associating a process with an interval %
| (| % TIMER on I_timer %
|   (| switch_display_mode := lr
|     | switch_light_on := ur
|     |) on I_timer
| % TIME_UPDATE on I_t_update %
|   (| change_item := ll
|     | update := lr
|     |) each I_t_update
| % STOPWATCH on I_stopwatch %
|   (| R := lr
|     | L := ur
|     |) on I_stopwatch
| % ALARM on I_alarm %
|   (| switch_chime := lr
|     | switch_alarm := ur
|     |) on I_alarm
| % ALARM_UPDATE on I_a_update expanded %
|   (| H := ] ll in MN, ll] init true
|     | 10MN := ] ll in H, ll]
|     | MN := ] ll in 10MN, ll]
|     | update_h := lr in H
|     | update_10mn := lr in 10MN
|     | update_mn := lr in MN
|     |) each I_a_update
|)
|)

```

Table 3: Control logic of the digital watch.

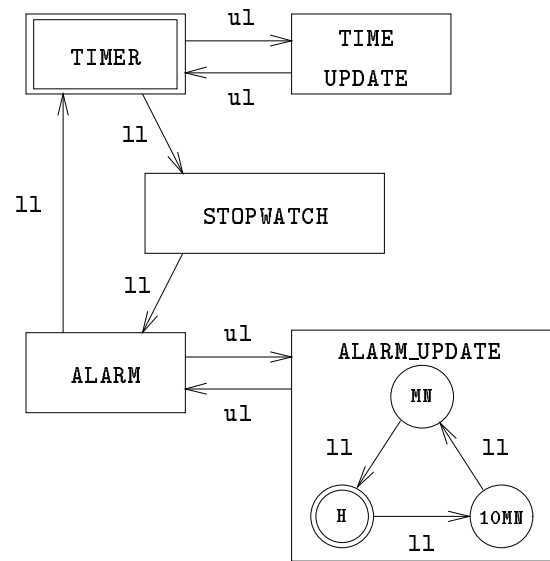


Figure 13: Automaton for the logic control of the digital watch.

- the `u1` button changes back to `TIMER` mode.
- the `1r` button updates the value (i.e., increments it), emitting event `update`.
- the `STOPWATCH` mode, where (see section 4.2):
  - the `11` button changes to `ALARM` mode,
  - the `u1` button has no effect,
  - the `1r` button is the `R` (`run`) button of the stopwatch,
  - the `ur` button is the `L` (`lap`) button of the stopwatch.
- the `ALARM` mode, where:
  - the `11` button changes to `TIMER` mode,
  - the `u1` button changes to `ALARM_UPDATE` mode,
  - the `1r` button switches the chime on and off, by emitting the event `switch_chime`,

- the `ur` button switches the alarm on and off, by emitting the event `switch_alarm`,
- the `ALARM_UPDATE` mode, where:
  - the `ll` button changes the updated component of alarm time, by switching between states: hour `H`, tens of minutes `10MN`, minutes `MN`,
  - the `ul` button changes back to `ALARM` mode,
  - the `lr` button updates the value (i.e., increments it) emitting, respectively, `update_h`, `update_10mn`, and `update_mn`.
- in any of the above modes, the `ur` button turns the alarm bell off.

An interval-based specification of this sequencing is given in table 3, where interval `I_timer` is initialized *inside* (i.e., `true`). The task `ALARM_UPDATE`, which is given an expanded form, compared to `TIME_UPDATE`, features sub-states for each component of the alarm time: this sub-automaton has initial state `H`, and is restarted in its initial state by the operator `each` on each entering the interval `I_a_update`.

The activity of the different sub-tasks in this example consists in transducing the incoming events (belonging to the set `{ll,lr,ul,ur}`), by interpreting them according to the current mode, into specific events acting on other parts of the watch. For example, in the `TIMER` task, the management of the display in twenty-four-hours or in a.m./p.m. mode corresponds to intervals:

```
(| 24H := ]switch_display_mode, switch_display_mode]
  | AM_PM := compl 24H
  |)
```

The same holds for the activity of the `ALARM` module, managing two state values, encoded in intervals;

```
(| CHIME := ]switch_chime, switch_chime]
  | ALARM := ]switch_alarm, switch_alarm]
  |)
```

These state informations must be permanently available, for displaying, comparing or computing purposes: hence they have to be *remanent* in the application.

## 6 Application to task sequencing in robotics

An application domain envisaged for the sequencing of data flow tasks is robotics. In this section we discuss possible guidelines for such applications.

**Task functions as data flow processes.** A robot task, at the relatively lowest level, consists of a *task function*, which is an equation  $c = f(s)$  giving the value of the command  $c$  to be applied to the actuator, in terms of the values  $s$  acquired by the sensors. The command to the actuator is a continuous function  $f$ , more or less complex, involving differential equations, and possibly dynamical aspects, referring to past values of  $s$  or  $c$ . Such a task can be composed of several sub-tasks, that can be given a certain priority order. An example of such a task is having a robot arm following the edge of an object: on the one hand, as a primary sub-task, it has to keep a constant distance from this object, as sensed e.g., by an infra-red sensor, and on the other hand, as a secondary sub-task, it must move at some constant speed. The implementation of such a task function is made by sampling sensor information  $s$  into a flow of values  $s_t$ , which are used to compute the flow of commands  $c_t$ :  $\forall t, c_t = f(s_t)$ . This kind of numerical, signal processing, data flow computation is the traditional application domain of data flow languages in general, and of SIGNAL in particular.

**Termination.** Such a process defines a behavior, but not a *termination*: this aspect must be defined separately. One way of deciding on termination of a robot task is to apply criteria for reaching a goal  $s_0$ : when this value is acquired by the sensor, the task is considered to have reached its goal, hence its end. An example is that of a trajectory tracking task, where the destination point constitutes the goal, and the criterium is the distance to this goal. In other terms, an error  $|s - s_0|$  has to be minimized, and the goal is reached with a precision  $\epsilon$  when condition  $|s - s_0| \leq \epsilon$  is satisfied. The evaluation of this condition must be performed at all instants: hence this evaluation is another data flow treatment. The moment when the condition is satisfied can be marked by a discrete event, which, causing termination of the task, can also cause a transition to another task at the higher level of the reactive sequencing. In this sense, this event can be used to specify the end of the execution interval of the task. Evaluation of such conditions can



be made following a dynamical evolution: a sequence of modes of evaluation of  $s$ , or of the criterium, can be defined, becoming finer (and possibly more costly) when nearing interesting or important values (an example of this in the stopwatch example is that when comparing alarm time to current time, minutes can be compared only once hours have been found to be equal, thus diminishing the comparisons).

**Sequencing.** At a somewhat higher level, a *robot task* can be defined as an already non-trivial behavior combining a task function, and various observers detecting, in the environment, events on which a reaction must be given: e.g., preconditions that should be verified before starting the task function, observers on conditions that could imply interrupting the robot task, or postconditions deciding on the task termination [10]. These conditions can be combinations of information from several different sensors, and evaluating such conjunctions of conditions, implies reasoning about their truth intervals and their overlapping. Reaction to the events corresponding to the observer interruptions can involve different treatment levels: inside the robot task (modifying parameters internally, without stopping), interrupting the task, and causing transition to another task, or interrupting the whole application, requesting external help. Finally, above the robot tasks level, an *application* level can be considered, where robot tasks are being sequenced according to various patterns. In particular, the composition function for output values of different sub-tasks described in section 5.2 can be used to combine effects of several tasks executed in parallel e.g., a movement combined with wall-following, the results of both tasks being “added” in order to form the resulting command.

**Application to robotic vision.** An application currently being worked on concerns the sequencing of active vision tasks specified under the form of a network of automata in SIGNAL [24]. The computations involved in the sensor-based command of an actuator correspond to our data flow processes. Determination of their start times and of their termination i.e., of their execution interval depends of events defined in terms of external inputs or internal state information: hence they can be described by tasks. The sequencing of vision tasks could then follow the patterns proposed earlier in this paper.

## 7 Related work

In this section, we briefly review related work on sequencing and data flow languages, mostly real-time and synchronous, from the perspective of their combination into a language-level common framework.

### 7.1 Sequencing languages

**Imperative languages.** Amongst the sequencing languages, an imperative one is ESTEREL [6], which is synchronous, meaning that its instructions are considered to be instantaneous in its formal model. It features various control structures, including parallelism, and three ways to stop a sub-process: natural termination (by reaching the end of a sequential statement), withdrawal (in an exception handling mechanism, where a last action on the instant of withdrawal is possible) and interruption (by a watchdog mechanism, cancelling any action in the interruption instant). Interruption and withdrawal are definitive: there is no way of restarting from the state where the stopping happened. In order to enable tasks with a duration to be executed, ESTEREL, in its recent version, supports an instruction `exec`, causing an external asynchronous process to be executed: it receives a start signal, emits a result signal, or can be stopped by a kill signal (no suspension and resuming are supported yet). The processes have to be managed by an external task controller [1], and are not written within the application, which compromises the use of the programming environments and analysis tools. An application of this construct is task-level programming in robotics [10].

The asynchronous language ELECTRE [28] provides a variety of interruption structures for tasks lasting over a duration. These interruptions can be grouped into numerous combinations. In addition to them, the events exchanged can be submitted to various memorization policies. The tasks can be suspended, and then resumed at their stopping state, or interrupted in a definitive way. Applications written in the language are compiled into automata, which can be submitted to various analysis and proof tools. When arriving at the lowest level of the hierarchy of tasks and sub-tasks in ELECTRE, the behavior of the primitive modules are external to the language.

**Graphical languages.** Classical graphical formalisms feature automata, but they are difficultly applicable as such to more than small problems, be-

cause of their lack of structure: a modification of a system specification might result in a very different automaton, which has to be completely revised, or even rewritten. Petri nets feature concurrency, which simplifies specification, and makes it more compact. Still, these place-transition nets are flat, and lack a hierarchy which would allow to factorize some aspects of a behavior at a high level: in this sense, they are still relatively low-level formalisms; besides, their semantics is not always clear [14]. Another graphical language, especially designed for and widely used in the control of physical processes, is GRAFCET (also called *Sequential Function Charts*); it consists of place-transition systems, controlling the activity of external tasks. Originally defined in a quite ambiguous fashion, GRAFCET is being studied in the sense of formalizing its semantics, and connecting it to the analysis tools from the synchronous approach [23].

A well known formally defined graphical language is STATECHARTS [15], which enables the graphical specification of hierarchical and parallel automata, with a variety of constructs amongst which the *history*, allowing for keeping of the last visited state of an suspended sub-automaton, and resuming of its activity in this current state. It has been influential as a specification language, along with the model defining its semantics. A recent extension, called hybrid STATECHARTS [21], concerns timed transition systems, where states are labeled by unconditional differential equations, thus specifying behaviors combining discrete and continuous evolutions i.e., hybrid systems [26, 29].

Another graphical language is ARGOS, belonging to the synchronous family, in a closer fashion than STATECHARTS, from which it is inspired [22]. It features the construction of automata, the possibility to encapsulate them inside a state of an upper-level automaton, in a hierarchical way, and a parallel composition of automata. It does not feature suspension or history. In its present version, it supports the specification of the control logic of applications, but not of actions or operations on values of events [14].

As a conclusion on sequencing languages, within our perspective, one can say that if they allow for hierarchical decomposition, down at the lowest level, the non-instantaneous, non-sequencing, continuous tasks that are sequenced are external to the language, hence to the programs written, and to the underlying model: there are two different, separated frameworks for the

sequencing on the one hand, and the tasks on the other hand. As was already noted above, drawbacks of this situation are that the analysis of applications is limited, and that homogeneity of the specification formalism is impaired.

Part of the motivation for the results presented in this paper is to integrate durational behaviors of tasks into the same framework as reactive transitions for sequencing.

## 7.2 Data flow languages

Data flow languages have originally been introduced as a declarative way, related to functional languages, of specifying algorithms for a particular organization of parallel architectures [12]. However, they can also be seen as a natural way of specifying certain types of computations, involving the repeated evaluation of output values from input values according to a given equation; in this sense, they are “very natural to control scientists” [14, p. 53], and “in digital signal processing, even without the motivation of concurrency” [17, p. 502]. Thus from this point of view, data flow languages seem to be the natural way of specifying non-sequencing, non-instantaneous tasks.

The data flow language LUSTRE [14], like SIGNAL, is synchronous. An important difference is that LUSTRE is functional, while SIGNAL is relational. The *definition principle* in LUSTRE says that “a variable is thoroughly defined from its declaration and the equation in the left part of which it appears” [14, p. 57]; in SIGNAL, this is not true for the clock definition, which can depend on the context in which the signal definition appears.

Neither of them does feature explicit sequencing constructs, but recent work on LUSTRE concerns recursive functions [9], which enable the introduction of a form of imperative sequencing e.g., re-initialization of a program when a condition becomes true.

It is possible to memorize state information in these data flow languages, using the delay operators; it is also possible to make computations conditioned by the presence of events or the value of boolean signals. Thus, it is possible to implicitly manage the sequencing of different modes of computation. But in general, programming sequencings that way is made unpractical by the absence of supporting structures, and the obligation to program them by hand.

### 7.3 Hybrids of sequencing and data flow

**Data flow networks of sequential processes.** In the data flow community, instances of hybrid sequential/data flow designs exist [12]: they concern the sequentiality of groups of operations in a data flow network. The sequential character of some parts of the computation in a data flow network can be given by the programmer, who supplies sequential processes as operators connected into a large-grain data flow network. It can also be extracted from a network of primitive operators, by determining “execution threads” that group some of them, and diminishing the amount of operators, hence of communications between operators, by implementing these sequential parts in the efficient “von Neumann” way.

In the synchronous approach, a combination of imperative and data flow features is made in  $\mu\text{GC}$  [5]: this formal model was defined in relation with the common formats for synchronous languages [27], and underlies the data flow approach to synchronous programming. In  $\mu\text{GC}$ , the imperative parts consist in ordering the actions inside an instant i.e., they are inside the operations in the data flow.

**Sequencing connection states of a data flow network.** Another way of combining sequencing and data flow is that of MANIFOLD [2], where an event mechanism is used for the sequencing of different connection states of a data flow network. However, the events and the units in the data flow are handled in an orthogonal fashion. The language is designed at a level inspired by hardware data flow architectures: is fully asynchronous, and features numerous instructions motivated by this system-level relevance. Thus, as a full-sized language, it is quite complex; an abstract model has been extracted from it.

As a conclusion, it can be noted that the relation between sequential and data flow computing seems to be made mostly at the level of the efficient implementation of data flow programs i.e., having sequential operations in a data flow network. The other way round, language-level sequencing of data flow tasks, is less current in the litterature.

The work presented here proposes a solution for this, motivated by applications in doamins concerning the discrete control of physical processes, e.g. robotics.

## 8 Perspectives

In this paper, we have presented how a concept of data flow task, and the sequencing of such tasks, could be built on top of the definition of time intervals in an instant-based data flow language like SIGNAL. The same approach should be applicable to other data-flow languages. This work is part of a global approach concerning the specification of dynamical behaviors of real-time systems, their verification and their synthesis. Perspectives for extensions of the work presented here are in the directions of:

**the compilation of intervals and sequencing patterns:** besides the compilation of the intervals themselves, and the synthesis of their clock, they provide us with an information that should be used for the analysis of the programs e.g., the detection of disjoint intervals, so that memory management can take advantage of it; the clock calculus could also benefit from this, for example in making use of the fact that signals filtered on disjoint intervals have disjoint clocks.

**the definition of operators for equational sequencing:** the patterns that have been proposed in sections 4 and 5 are exploratory work on the way to more abstract, less imperative patterns, that are fully interval-based: operators like union, intersection, complement, ..., and relations like disjunction, inclusion, ... will be used to specify relations between series of interval occurrences, which, when duly constrained, constitute an algorithm. The goal is to be able to specify the dynamical behavior of a reactive system by a system of equations on the time intervals, and to solve it in the SIGNAL equational framework.

In the perspective of the specification of dynamical behaviors and properties, time intervals are also related to the verification and synthesis of reactive systems, or to hybrid systems [4, 26, 29, 33].

**the definition of a three-valued interval calculus:** as is suggested by the clock calculus defined on  $\mathbb{Z}/3\mathbb{Z}$ , briefly presented in section 3.1.4, and by the discussion on suspendable tasks in section 2.4.1, interesting states of an activity or a task are: active, inactive but present, and

absent. If encoding these three states as:

0	if absent (terminated / not started)
+1	if active
-1	if inactive (suspended)
$\pm 1$	if present

behaviors can be specified equationally e.g., for intervals (or tasks)  $i$  and  $j$ ,  $i^2 + j^2 = 1$  means that one of them is +1 while the other one is 0 i.e., one of them is absent and the other is present i.e., finally, that they are complementary. The equation  $i^2 + j^2 = k^2$  means the same, but relatively to interval  $k$ .

**applications to:**

- task-level programming in robotics, with applications in active robotic vision [24], from the level of data flow command, up to sequencing of tasks. General structures should be identified, that facilitate the high-level programming of robotic applications, taking into account their specific aspects: reactivity, complementarity between “continuous”, numerical computations, and discrete event directed dynamical behavior. This can also be related to works on the definition of generic robot controllers [10, 32], and to task-level robot programming models [31].
- execution models for planning in artificial intelligence, especially reactive planning, or more accurately reactive plans [34], and temporal planning [30]: an application of this can be the execution of high-level reactive plans on robotic architectures; it relates to the definition of hierarchical architectures for autonomous systems [25], and the automatic synthesis of sequencings [16].

## References

- [1] C. André, J.P. Marmorat, J.P. Paris. Execution machines for ESTEREL. In *Proc. of the European Control Conference, ECC'91*, Grenoble, France, July, 1991.

- 
- [2] F. Arbab, E. Rutten. MANIFOLD: a programming model for massive parallelism. In *Proc. of the Working Conference on Massively Parallel Programming Models, MPPM'93*, Berlin, Germany, September 20-23, 1993.
  - [3] A. Benveniste, G. Berry. The synchronous approach to reactive and real-time systems. *Another look at real-time programming*, special section of *Proceedings of the IEEE*, 79(9), September 1991.
  - [4] A. Benveniste, M. Le Borgne, P. Le Guernic. Signal as a model for real-time and hybrid systems. In *Proceedings of the 4<sup>th</sup> European Symposium on Programming, ESOP '92*, Rennes, France, LNCS n° 582, Springer, 1992.
  - [5] A. Benveniste, P. Le Guernic, P. Caspi, N. Halbwachs. Data flow synchronous languages. In *Proceedings of the REX '93 School/Workshop on Concurrency*, Noordwijkerhout, The Netherlands, June 1-4, 1993.
  - [6] G. Berry, G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2), November 1992.
  - [7] P. Bournai, P. Le Guernic. *Un environnement graphique pour SIGNAL*. IRISA/INRIA-Rennes, Research Report, n° 741, July 1993. (In French.)
  - [8] P. Bournai, B. Chéron, T. Gautier, B. Houssais, P. Le Guernic. *SIGNAL manual*. IRISA/INRIA-Rennes, Research Report, n° 745, July 1993.
  - [9] P. Caspi. *Introduction de la récursivité en flots de données synchrones*. IMAG, Laboratoire de Génie Informatique, Rapport Technique, n° SPECTRE L-18, November 1992. (In French.)
  - [10] E. Coste-Manière, B. Espiau, E. Rutten. A task-level robot programming language and its reactive execution. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, Nice, France, May 12-14, 1992.
  - [11] B. Dutertre. *Spécification et preuve de systèmes dynamiques*. Ph.D. Thesis, IFSIC, University of Rennes I, France, December 1992. (In French.)



- [12] J.L. Gaudiot, L. Bic eds.. *Advanced topics in data flow computing*. Prentice Hall, 1991.
- [13] T. Gautier. *Conception d'un langage flot de données pour le temps réel*. Ph.D. Thesis, Computer Science Dept., University of Rennes I, France, December 1984. (In French.)
- [14] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.
- [15] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), pp. 231–274, 1987.
- [16] I. Klein. *Automatic synthesis of sequential control schemes*. Ph.D. Thesis, Dept. of Electrical Engineering, University of Linköping, Sweden, 1993.
- [17] E.A. Lee. Static scheduling of data flow programs for DSP. In [12], pp. 501–526.
- [18] 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 *Proceedings of the IEEE*, 79(9), September 1991.
- [19] O. Maffeis, P. Le Guernic. Combining dependability with architectural adaptability by means of the SIGNAL language. *Proceedings of the 3<sup>rd</sup> Workshop on Static Analysis*, Padova, Italy, LNCS n° 724, Springer, 1993.
- [20] Z. Manna, A. Pnueli. *The Temporal logic of reactive and concurrent systems : specification*. Springer, 1992.
- [21] Z. Manna, A. Pnueli. *Models for reactivity*. Stanford University, Department of Computer science, Research Report, n° STAN-CS-92-1461, January 1993.
- [22] F. Maraninchi. Argonaute: graphical description, semantics and verification of reactive systems by using a process algebra. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, LNCS n° 407, Springer, 1989.

- 
- [23] L. Marcé, P. Le Parc. Defining the semantics of languages for programmable controllers with synchronous processes. *Control Engineering Practice*, 1(1), February 1993. (also in *Proceedings of the 18<sup>th</sup> IFAC Workshop on Real-Time Programming*, Bruges, Belgium, June 1992)
  - [24] E. Marchand, F. Chaumette, E. Rutten. *Stratégie perceptive d'un environnement statique dans un contexte de vision active*. IRISA/INRIA-Rennes, Research Report, n° 775, Octobre 1993. (In French.)
  - [25] M. Morin, S. Nadjm-Tehrani, P. Österling, E. Sandewall. Real-time hierarchical control. *IEEE Software*, September 1992.
  - [26] X. Nicollin, J. Sifakis, S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18(9), September 1992.
  - [27] J.P. Paris, G. Bery, F. Mignard, P. Couronné, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, F. Dupont, C. Le Maire. *Projet SYNCHRONE – Les formats communs des langages synchrones*. INRIA, Technical Report, n° 157, June 1993. (In French.)
  - [28] J. Perraud, O. Roux, M. Huon. Operational semantics of a kernel of the language ELECTRE. *Science of Computer Programming*, 97(1), pp. 83–103, 1992.
  - [29] A. Ravn, H. Rischel, K.M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1), January 1993.
  - [30] E. Rutten, J. Hertzberg. Temporal planner = nonlinear planner + time map manager. *A.I. Communications*, 6(1), pp. 18–26, March 1993.
  - [31] E. Rutten, L. Marcé. An imperative language for task-level programming: definition in temporal logic. *Artificial Intelligence in Engineering*, 8(4), to appear, 1993.
  - [32] D. Simon, B. Espiau, E. Castillo, K. Kapellos. *Computer-aided design of a generic robot controller handling reactivity and real-time control issues*. INRIA - Sophia Antipolis, Research Report, n° 1801, November 1992.

- [33] R. K. Shyamasundar. Specification of hybrid systems in CRP. *Proceedings of the International Conference on Algebraic Methodology and Software Technology, AMAST '93*, Twente, The Netherlands, June 21 – 25, 1993.
- [34] S. Thiébaux, J. Hertzberg. A Semi-Reactive Planner Based on a Possible Models Action Formalization. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems (AIPS92)*, 228-235, Morgan Kaufmann, 1992.
- [35] K. Wolinski, M. Belhadj. *Vers la synthèse automatique de programmes SIGNAL*. IRISA/INRIA-Rennes, Research Report, n° 746, July 1993. (In French.)

## Contents

<b>1</b>	<b>Motivation</b>	<b>1</b>
1.1	Problem addressed . . . . .	1
1.2	Context . . . . .	3
1.3	Proposed approach . . . . .	4
<b>2</b>	<b>Tasks: associating a process with a time interval</b>	<b>6</b>
2.1	Applications and processes . . . . .	6
2.2	Time intervals . . . . .	6
2.3	Tasks . . . . .	8
2.4	Derived task behaviors . . . . .	10
2.4.1	Suspending and terminating a task . . . . .	10
2.4.2	Suspending activity, with values available . . . . .	11
2.4.3	Restarting anew on each beginning event . . . . .	11
<b>3</b>	<b>SIGNAL, time intervals and tasks</b>	<b>13</b>
3.1	A brief introduction to SIGNAL . . . . .	13
3.1.1	An equational synchronized data flow language . . . . .	13
3.1.2	The kernel of SIGNAL . . . . .	14
3.1.3	Modularity and derived processes . . . . .	15
3.1.4	Compilation and clock calculus . . . . .	16
3.2	Time intervals in SIGNAL . . . . .	17
3.3	The encoding of tasks in SIGNAL . . . . .	19
3.3.1	Primitive processes level . . . . .	19
3.3.2	Task process level . . . . .	21
<b>4</b>	<b>Sequencing time intervals</b>	<b>23</b>
4.1	Place-transition networks . . . . .	23
4.2	Example: the control logic of a stopwatch . . . . .	26
4.3	Particular sequencing patterns . . . . .	27
<b>5</b>	<b>Sequencing data flow tasks</b>	<b>29</b>
5.1	Process composition . . . . .	29
5.2	Modularity and hierarchy . . . . .	30
<b>6</b>	<b>Application to task sequencing in robotics</b>	<b>35</b>

<b>7</b>	<b>Related work</b>	<b>37</b>
7.1	Sequencing languages . . . . .	37
7.2	Data flow languages . . . . .	39
7.3	Hybrids of sequencing and data flow . . . . .	40
<b>8</b>	<b>Perspectives</b>	<b>41</b>

## List of Figures

1	Phases in the segmentation of an acoustic speech signal. . . .	1
2	Decomposing $]\alpha, \omega]$ into sub-intervals. . . . .	7
3	Counter suspended on $]\mathbf{S}, \mathbf{R}]$ (a), each $]\mathbf{B}, \mathbf{E}]$ (b). . . . .	10
4	The time interval programmed with the graphical interface. . .	18
5	A possible trace of a time interval and its bounding events. . .	18
6	The time interval as a two-state automaton. . . . .	19
7	Filtering input and output of $\mathbf{P}$ on $\mathbf{I}$ : example of an interval. .	21
8	A transition between two states. . . . .	24
9	A place with incoming and outgoing transitions. . . . .	24
10	Automaton for the logic control of a stopwatch. . . . .	26
11	A task $\mathbf{P}$ behaves like $\mathbf{P1}$ until event $\mathbf{E}$ , and then like $\mathbf{P2}$ . . . .	29
12	A process $\mathbf{P}$ with sub-tasks $\mathbf{P1}$ and $\mathbf{P2}$ . . . . .	31
13	Automaton for the logic control of the digital watch. . . . .	33

## List of Tables

1	Primitives-level encodings of <b>on</b> and <b>each</b> . . . . .	20
2	Encodings of $\mathbf{I}$ on $\mathbf{J}$ and $\mathbf{I}$ each $\mathbf{J}$ . . . . .	22
3	Control logic of the digital watch. . . . .	32



---

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,  
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY  
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex  
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1  
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex  
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

---

Éditeur  
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)  
ISSN 0249-6399