The sequencing of data flow tasks in SIGNAL: application to active vision in robotics [1]

Eric Rutten *, Eric Marchand ** and François Chaumette **
IRISA / INRIA - Rennes
F-35042 RENNES, France
fax: (+33) 99.38.38.32, e-mail: ⟨*name*⟩@irisa.fr

Abstract

Many applications of real time systems feature a combination of "continuous" (possibly sampled) and discrete (sequencing and task control) behaviors, both reactive to the evolutions of the execution environment. In particular, robotics applications involve the programming of the control functions for each task, and the sequencing of such tasks at a higher level.

In the framework of the real time synchronized data-flow language SIGNAL, *we propose extensions based on* time intervals, *enabling the specification of* tasks, *and of their suspension or abortion for their* sequencing *in reaction to discrete events. This paper illustrates the application of these constructs to* active vision in robotics, *where successive phases have to be sequenced, each associated with a particular control law, in order to recognize a complex environment.*

1 Reactivity and robot programing

Reactivity is an essential aspect of real time systems, and the analysis of their safety requires an underlying formal model. Synchronous languages aim at providing a complete framework for the specification, verification and execution of such real time applications [7]. Among them, SIGNAL is a real-time synchronized data-flow language [10]. Its model of time is based on instants, and its actions are performed within the instants. However, various application domains such as signal processing and robotics require the possibility of specifying behaviors composed of the succession of different modes of durative interaction with their environment.

To this purpose, we introduce the notion of *time interval*, defined by a start and an end event [11]. Associating a time interval to a data-flow process specifies a *task* i.e., a non-instantaneous activity and its execution interval. Hence, it is possible to specify the *sequencing of data-flow tasks*, declaratively, themselves possibly involving a sequencing of sub-tasks, by constraining their intervals. In particular, it is possible to specify hierarchical, parallel automata. Both the data flow and the sequencing aspects are in the *same language* framework, thus relying on the *same model* for their execution and the verification of the correctness of programs.

Task-level programming of robots consists in specifying robot tasks and sequencing them i.e., associating them with modes on which they are enabled [4]. A robot task is a data flow function computing the flow of values of control to the actuator from the flow of sensor input data. For example, movements toward some point can alternate with prehension tasks, or assembly of objects. The transition between the various execution modes is driven reactively by the reception of externally sensed, or internal information [3].

Our task structure applies well to robotics by encoding the data flow task functions as well as their hierarchical sequencing within the same framework as shown by an application to a perceptive strategy of a static environment in an active vision context [9].

Related work. Robot programming involves the discrete event driven sequencing of robot tasks implementing a continuous task control function [4]. This duality between immediate reaction to events and durative execution of a task function can be handled by a synchronous/asynchronous approach [3] where the behavior is encoded in a synchronous automaton, while asynchronous external processes implement durational control laws. However, the disadvantage is that the

[1] This work was partly supported by the CNRS inter-PRC project VIA (*Vision Intentionnelle et Action*).

*EP-ATR Project (Programming Environment for Real-Time Applications).

**TEMIS Project (Image Sequence Analysis).

two aspects are developed in separate languages and models, thus making analysis of interactions difficult. Synchronous programming languages are either imperative or declarative (data flow) [7].

The advantage of the constructs we proposed is that they enable both the specification of the functions relating input and output values (data flow), and that of the transition between different such behaviors (sequencing), *within the same programming language* framework. An approach related to ours integrates ARGOS (hierarchical parallel automata) with LUSTRE (data flow) [8]; we try to specify sequencing in a more declarative style.

2 Data-flow, intervals and tasks

Data flow applications are activities executed over a set of instants in time i.e., non-instantaneously: at each instant, data is acquired from the execution environment, and processed along a network of operators, in order to produce output values.

SIGNAL [10] is a synchronous real-time language, data flow oriented (i.e., declarative) and built around a minimal kernel. 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 $(\mathtt{x}_t)_{t \in T}$ 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 the program is constrained so as to compute a deterministic solution, then executable code is automatically produced (in C or FORTRAN). The complete programming environment also features a graphical, block-diagram oriented user interface, a prover for dynamical properties of programs, and work is in progress concerning synthesis of integrated circuits.

The kernel comprises the five following primitives. *Functions* are defined on the types of the language (e.g., boolean negation of signal `E`: **`not`** `E`). The signal (Y_t), defined by the instantaneous function f in: $\forall t,\ Y_t = f(X_{1_t}, X_{2_t}, ..., X_{n_t})$ is encoded in SIGNAL by: `Y := f{ X1, X2, ... , Xn}`. The signals `Y`, `X1`, ..., `Xn` are required to have the same clock. *Delay* gives the past value of a signal $ZX_t = X_{t-1}$, with initial value V_0: `ZX := X$1` with initialization `ZX init V0`; `X` and `ZX` have the same clock. *Selection* of a signal `X` according to a boolean condition `C` is: `Y := X when C`; 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`**; when `Y` is present, its value is that of `X`. *Deterministic merge* defines the union of two signals of the same type: `Z := X default Y`. The clock of `Z` is the union of that of `X` and that of `Y`. 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. *Parallel composition* of processes is made by 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: `(| P1 | P2 |)`. For example equation $x_t = x_{t-1} + 1$ is also $x_t = zx_t + 1, zx_t = x_{t-1}$ i.e. 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`.

The rest of the language is built upon this kernel. A structuring mechanism is proposed in the form of process schemes, defined by a name, typed parameters, input and output signals, a body, and local declarations. Occurrences of process schemes in a program are expanded by a pre-processor of the compiler. Derived processes have been defined from the primitive operators, providing programming comfort, such as: **`synchro`**`{X,Y}` which specifies the synchronization of signals `X` and `Y` and `X` **`cell`** `B` which memorizes (using a delay) values of `X` and outputs them also when `B` is true. Arrays of signals and of processes have been introduced as well. An example is the simple counter in Figure 1, where `X` is the number of occurrences of **`UNIT`**

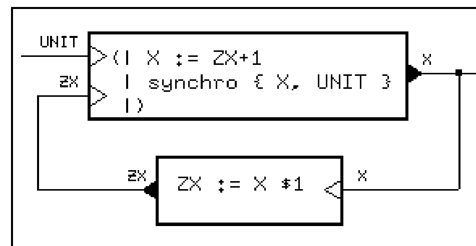

Figure 1: *A counter in* SIGNAL.

(as X is incremented by 1 each time it is present, and it is present each Unit is present), and ZX is initially 0.

A data flow application is executed from an initial state of it memories at an initial instant α which is before the first event of the reactive execution. A data flow process has no termination specified in itself: therefore its end at instant ω can only be decided in reaction to external events or the reaching of given values. Hence ω is part of the execution, and the time interval on which the application executes is the left-open, right-closed interval $]\alpha,\omega]$.

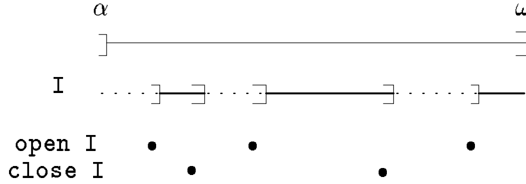

Figure 2: *Decomposing* $]\alpha,\omega]$ *into sub-intervals.*

Time intervals are introduced in order to enable the structured decomposition of the interval $]\alpha,\omega]$ into sub-intervals as illustrated in fig. 2, and their association with processes [11]. Such a sub-interval I is delimited by occurrences of bounding events at the beginning B and end E: `I := ]B, E]`. It has the value **inside** between the next occurrence of B and the next occurrence of E, and **outside** otherwise. Like $]\alpha,\omega]$, sub-intervals are left-open and right-closed. This choice is coherent with the behavior expected from 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.

The operator `compl I` defines the complement of an interval I, which is **inside** when I is **outside** and reciprocally. Operators `open I` and `close I` respectively give the opening and closing occurrences of the bounding events. Occurrences of a signal X inside interval I can be selected by `X in I`, and reciprocally outside by `X out I`. In this framework, **open** I is B **out** I, and **close** I is E **in** I.

Tasks consist in associating some (sub)process of the application with some (sub)interval of $]\alpha,\omega]$ on which it is executed. Tasks active on $]\alpha,\omega]$ 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. Tasks are defined by the process P executed, the execution interval I, and the starting state (current, or initial) when (re-)entering the interval.

More precisely, the latter means that, when re-entering the task interval, the process can be started at its current state at the instant where the task was *suspended* (meaning: in a temporary fashion): this is written `P on I`. Alternately, it can be started at its initial state as defined by the declarations of all its state variables, if the task was *interrupted* (meaning: aborted in a definitive fashion): `P each I`. For example, the counter of Figure 1 can be transformed into a simple stopwatch, that runs from a pressure on a button R, until the next pressure on the same button, by: **COUNT{UNIT} on]R,R]**. This simply specifies that the counting behaviour is confined on the interval between occurrences of event R.

The processes associated with intervals can themselves be decomposed into sub-tasks: this way, the specification of *hierarchies* of complex behaviors is possible. For example, the simple stopwatch given above can be re-used into a resetable stopwatch as follows: **(COUNT{UNIT} on]R,R]) each]R,Stop]**, where]R,R] is initially **inside**. The simple stopwatch is then re-initialized on each entering]R,Stop].

Task control is achieved as a result of constraining bounding events of intervals, and associating activities to them, either to be suspended or aborted. *Parallelism* between several tasks is obtained naturally when tasks share the same interval, or overlapping intervals. *Sequencing tasks* then amounts to constraining the intervals of the tasks. Using **on** and **each** as above already enables controlling activities; more elaborate behaviors can be specified as follows. This way, it is possible to specify hierarchical parallel automata or place/transition systems.

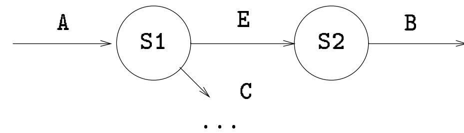

Figure 3: *Transitions between states.*

Each time interval holds some state information, and events cause transitions between these states. In the simple *timeout* behavior illustrated in Figure 3, a transition leads from state `S1` to state `S2` on the occurrence of an event `E`, except if the event `C` occurs before. This can be coded by two intervals such that the closing of the one on the occurrence of event `E` is the opening of the other one, as follows:

```
(| S1 := ]A , E default C]
 | S2 := ]E in S1, B] |)
```

An encoding of intervals and tasks into the SIGNAL kernel exists [11].

3 Application to active robotic vision

The sequencing of synchronous data flow tasks has been applied to a robot vision problem : the 3-D structure estimation of a set of geometrical primitives in an active vision context.

Active robotic vision The observability of the camera motion which is necessary for the structure estimation characterizes a domain of research called dynamic vision [5]. When the camera motion is controlled using vision data, dynamic vision becomes active vision [1][12] which generally provides more precise results. Such camera motions are performed using the visual servoing approach *i.e*, using a control law in closed loop with respect to vision data [6].

The aim of this scheme is to obtain a complete and precise description of a 3-D scene using the visual data provided by a camera mounted on the end effector of a robot arm. In particular, we are interested in the reconstruction of a nuclear plant environment, constituted by a set of cylinders.

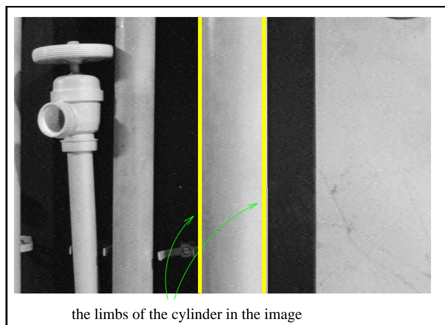

Figure 4: *Cylinder in a nuclear plant environment (the white straight lines represent the limbs of the cylinder)*

In order to obtain a precise estimation of the structure of a selected primitive, we have shown that it is necessary that the camera focus on it and realize particular motion [2]. For example, a cylinder must always appear centered and horizontal or vertical in the image sequence (see figure 4) and the camera has to turn around it in order to obtain a non-biased and robust estimation of its radius and spatial localisation. This *optimal estimation* process can be divided into 3 parts (see figure 5) : an estimation based on only one of the two limbs of the cylinder is performed ($\mathcal{E}_{e_1}$). It allows us to compute the position of the second limb in order to get a better estimation based on the two limbs ($\mathcal{E}_{e_2}$). The last step consists in acquiring the length of the cylinder ($\mathcal{E}_l$).

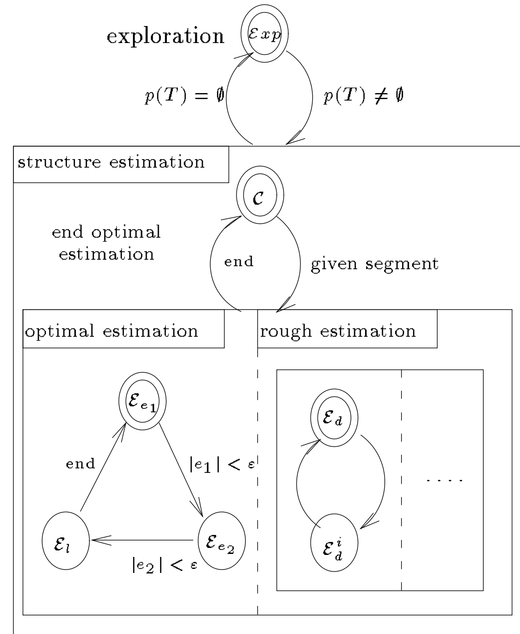

Figure 5: *Automata network*

In order to obtain a complete estimation of the scene, the optimal estimation process has to be successively realized for each primitive of the scene. So we have developped a method for connecting up several estimations [9]. It is based on image data and on a rough estimation of the structure of the other primitives. For a given position T of the camera, a data base $p(T)$ containing the observed segments is created. These segments fit with the limbs of the cylinders. A *selection* process ($\mathcal{C}$) chooses one of the segments, then the camera focuses on it and the *optimal estimation* process provides a robust estimation of the parameters of the corresponding cylinder.

At the same time, a *rough estimation* ($\mathcal{E}_r$) of the structure of the others primitives is done (rough since the camera motion is not optimal for these other primitives). It gives us information about the scene which will be used in an *exploration* process. Every *optimal estimation* ends when all the cylinder parameters have been computed. The data base is then updated and a new segment is chosen ($\mathcal{C}$). When all the data base elements have been treated, an *exploration* process ($\mathcal{E}xp$) is required in order to ensure that the whole scene has been estimated.

Application Programming. This method has been implemented with an automata network using the real time language SIGNAL and the notion of time interval defined above. These automata are able to connect up the different stages of the reconstruction process (selection, focusing, optimal estimation of the selected primitive and concurrently, rough estimation of the other ones) and to provide a robust estimation of the spatial organisation of the scene. These automata network, which manage the tasks sequencing, and the estimation tasks has been written in SIGNAL using the sequencing of data flow tasks describe above.

At this step of the description, we provide a part of the SIGNAL code to illustrate the feasability of encoding such an automata network with our approach based on time interval description (cf table 1).

Let us now examine the interests in using the SIGNAL language for a robot vision application :

- with synchronous languages, a mathematically well defined semantics is at basis of the language implementation. When compiling, SIGNAL code is translated into a graph on which correctness proofs can be performed and dynamical properties can be proved ;
- SIGNAL is an equational data flow language and, as we said above, our estimation tasks are performed using control laws in closed loop with respect to vision data. The implementation of such a loop is very easy to express in SIGNAL. Moreover, the data flow structure of the estimation algorithm which uses at any time t the information given by the camera and the parameters estimation at time $t-1$ matches to the language philosophy and the delay operator ;
- when the automata network specification is done, programming such an automata is quite easy when using the time intervals describes above. The source code, in SIGNAL, of the application is very close to the specification because programming is performed via the specification of constraints or relations between all the involved signals.
- specification of the time interval corresponding to the rough estimation tasks, which must be performed in parallel with the optimal estimation tasks, does not raise any problem (except the fact that dynamic creation of a new process is not possible). We have here a parallelism of specification, and the compiler manages all the synchronisation and communication problems.

```
(| I_E := ] when p(T) = ∅, when p(T) ≠ ∅ ]
                                 init  inside
 | I_REC := compl I_E
 | Exploration each I_E
 | Structure_estimation each I_REC  |)
```

```
Structure_estimation
(| I_C := ] length_performed, Segment_choosen ]
                                 init inside
 | I_R := compl I_C
 | New_Choice each I_C
 | Primitive_estimation each I_R  |)
```

```
Primitive_estimation
(| Optimal_estimation
 |(|Rough_estimation_1|...|Rough_estimation_n|)|)
```

```
Rough_estimation_i
(| I_Rs := ] (Find_new_segment in I_R),
                     (Segment_lost in I_R) ]
 | Rough_estimation each I_Rs |)
```

```
Optimal estimation
(| (| I_Rp := ] close R_l , Accuracy_reached1 ]
    | Accuracy_reached1 := when |prec|< ε |)
 | (|  I_Rv := ] close I_Rp , Accuracy_reached2 ]
    | Accuracy_reached2 := when |prec|< ε' |)
 | (| I_Rl := ] close I_Rv , length_performed ]
    | length_performed := when g(p(T_φ)) |)  |)
```

Table 1: *Program for the application*

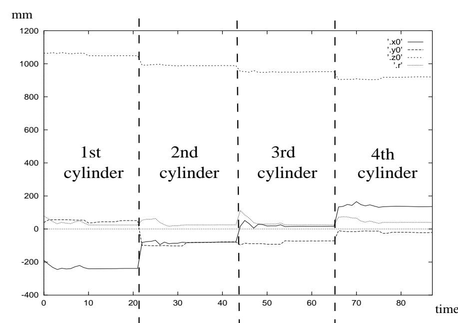

Figure 6: *Estimated position parameters and radius estimation of a set of 3 cylinders*

Experimental Results Figure 6 shows the result of the 3-D reconstruction of a scene made up with 4 cylinders. The experimental results depicted in figure 6 show the position parameters of the cylinder (x_0, y_0, z_0) and the radius r computed at each iteration and expressed in the initial camera frame. This real experiment has been done on a an experimental testbed constituted by a CCD camera mounted on the end effector of a 6 dof robot. The sequencing of the estimation tasks is written with SIGNAL, but the reconstruction tasks themselves are not yet implemented with this language, even if they have been implemented in simulation to show the feasability of programming such tasks with SIGNAL.

Figure 7 shows the results of the rough estimation of the second cylinder parameters in parallel with an optimal reconstruction of the first one, then the optimal reconstruction of the second cylinder: note that the rough estimation (see figure 7) is far worse than the optimal estimation. It underlines the fact that active vision can significantly improve the estimation accuracy.

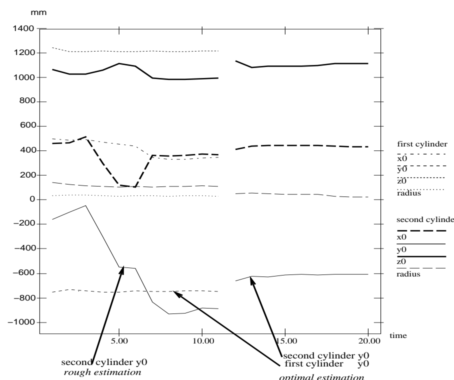

Figure 7: *Rough estimation of the second cylinder parameters in parallel with an optimal reconstruction of the first one, then the optimal reconstruction of the second cylinder*

4 Conclusion

We presented a language-level integration of the data flow and sequencing paradigms, and its application to the sequencing of robot tasks in active vision. It is specified in terms of time intervals in the framework of an instant-based synchronized data flow language: SIGNAL. Its constructs enable the designation of time intervals, their association with data flow processes in order to form tasks, and the sequencing of these data flow tasks. This work is part of a global approach concerning the specification of dynamical behaviors of real-time systems.

The application of these constructs to robotic vision demonstrates its adequacy for task-level robot programming. Tasks appear to be particularly fitting to the data flow nature of the control functions between sensor data and control output, while constraining their execution intervals makes it possible to specify the sequencing of robotic tasks.

References

[1] J. Aloimonos. Purposive and qualitative active vision. *Proc. on* 10th *ICPR*, New Jersey, 1990.

[2] F. Chaumette, S. Boukir. Structure from motion using an active vision paradigm. *Proc. on* 11th *ICPR*, s The Hague, Holland, September 1992.

[3] E. Coste-Manière. A synchronous/asynchronous approach to robot programming. In *Proc. of the EUROMICRO Workshop on Real-Time Systems*, Oulu, Finland, June 22–24, 1993.

[4] E. Coste-Manière, B. Espiau, E. Rutten. A task-level robot programming language and its reactive execution. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Nice, France, May 12–14, 1992.

[5] J.L Crowley, P. Stelmaszik, P. Puget. Measurement and integration of 3-D structures by tracking edge lines. *Int. Journal of Computer Vision*, Vol. 8, No. 1, July 1992.

[6] B. Espiau, F. Chaumette, P. Rives. A new approach to visual servoing in robotics. *IEEE Trans. on Robotics and Automation*, Vol 8, No. 3, June 1992.

[7] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.

[8] M. Jourdan, F. Lagnier, F. Marananchi, F. Raymond. A multiparadigm language for reactive systems. In *Proc. of the IEEE Int. Conf. on Computer Languages, ICCL '94*, Toulouse, France, May, 1994.

[9] E. Marchand, F. Chaumette, E. Rutten. *Stratégie perceptive d'un environnement statique dans un contexte de vision active.* INRIA Research Report, no. 2092, Octobre 1993. (Anonymous FTP : `ftp.inria.fr`, `INRIA/publication/RR/RR-2092.ps.Z`) (In French)

[10] 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.

[11] E. Rutten, P. Le Guernic. *Sequencing data flow tasks in* SIGNAL. In *Proc. of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, Florida, June 21, 1994.

[12] G. Sandini,M. Tistarelli. Active tracking strategy for monocular depth inference over multiple frames. *IEEE Trans. on PAMI*, Vol. 12, No. 1, January 1990.