

The Synchronous approach to reactive and real-time systems

Albert Benveniste, Gérard Berry

▶ **To cite this version:**

Albert Benveniste, Gérard Berry. The Synchronous approach to reactive and real-time systems. [Research Report] RR-1445, INRIA. 1991. inria-00075115

HAL Id: inria-00075115
https://inria.hal.science/inria-00075115

Submitted on 24 May 2006

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

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

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

1992

2ème anniversaire

N° 1445

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

THE SYNCHRONOUS APPROACH TO REACTIVE AND REAL-TIME SYSTEMS

Albert BENVENISTE
Gérard BERRY

Juin 1991

RR_1445

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX
FRANCE
Téléphone : 99.36.20.00
Télex : UNIRISA 950 473F
Télécopie : 99.38.38.32

The Synchronous Approach to Reactive and Real-Time Systems

Albert BENVENISTE, benveniste@irisa.fr
IRISA-INRIA, Campus de Beaulieu,
35042 RENNES CEDEX, FRANCE

Gérard BERRY, berry@cma.cma.fr
Ecole des Mines, Centre de Mathématiques Appliquées
Sophia-Antipolis, 06560 VALBONNE, FRANCE

April 5, 1991

Publication Interne n° 581 avril 1991 (36 pages) - Programme II

Abstract

This special issue[1] is devoted to the *synchronous* approach to reactive and real-time programming. This introductory paper presents and discusses the application fields and the principles of synchronous programming. The major concern of the synchronous approach is to base synchronous programming languages on mathematical models. This makes it possible to handle compilation, logical correctness proofs, and verifications of real-time programs in a formal way, leading to a clean and precise methodology for design and programming.

L'approche synchrone pour les systèmes réactifs et temps-réels

Résumé

Ce papier est une introduction informelle aux principes de la programmation synchrone. On y discute les domaines d'application de ce concept. On en présente les principes selon les deux points de vue "système de transition à états" et " systèmes d'équations dynamiques". Puis on étudie les liens entre points de vue synchrone et asynchrone.

[1] this paper is to appear in the special issue of the Proceedings of the IEEE entitled *Another Look at Real-Time Programming*

The Synchronous Approach to Reactive and Real-Time Systems

Albert BENVENISTE (FELLOW, IEEE), benveniste@irisa.fr
IRISA-INRIA, Campus de Beaulieu,
35042 RENNES CEDEX, FRANCE

Gérard BERRY, berry@cma.cma.fr
Ecole des Mines, Centre de Mathématiques Appliquées
Sophia-Antipolis, 06560 VALBONNE, FRANCE

March 22, 1991

Abstract

This special issue is devoted to the *synchronous* approach to reactive and real-time programming. This introductory paper presents and discusses the application fields and the principles of synchronous programming. The major concern of the synchronous approach is to base synchronous programming languages on mathematical models. This makes it possible to handle compilation, logical correctness proofs, and verifications of real-time programs in a formal way, leading to a clean and precise methodology for design and programming.

1 Introduction: Real-Time and Reactive Systems

It is commonly accepted to call *real-time* a program or system that receives external interrupts or read sensors connected to the physical world and outputs commands to it. Real-time programming is an essential industrial activity whose importance keeps increasing. Factories, plants, transportation systems, cars, and a wide variety of everydays objects are or will be computer controlled.

However, there is still little agreement about what the precise definition of a real-time system should be. Here, we propose to call *reactive* a system that maintains a permanent interaction with its environment[1] and to reserve

[1] The notion of a reactive system was first introduced in [14, 23].

the word *real-time* for reactive systems that are in addition subjet to externally defined *timing constraints*. The broad class of reactive applications therefore contains all real-time applications as well as non-real-time applications such as classical communication protocols, man-machine interfaces, etc.

Safety is a crucial concern for reactive and real-time programs. In this area, a simple bug can have extreme consequences. *Logical correctness* is the respect of the input/output specification; it is essential in all cases. *Temporal correctness* is a further requirement of real-time applications: a logically correct real-time program can fail to adequately control its environment if its outputs are not produced on time. Notice that the expressions "timing constraints" and "on time" should not be taken too literally, since constraints are not necessarily expressed in terms of physical time; for example, "stop in less than 30 meter" is a timing constraint expressed by a distance.

Historically, reactive and real-time applications mostly evolved from the use of analog machines and relay circuits to the use of microprocessors and computers. They did not benefit from the recent progress in programming technology as much as did other fields. Although strongly technically related, the various application fields are treated by different groups of people having their own methods and vocabulary, and little relation has been established between them. The programming tools are still often low-level and specific. For instance, one uses calls to specific operating systems to monitor the communications between modules written in standard languages such as assembly or C, and one writes non-portable programs designed to drive very specific hardware units.

The present situation must change rapidly. Modern applications will require strong interactions between application fields that used to be separated, and specific vocabularies or tools must be unified whenever possible to keep large systems tractable. Low-level programming techniques will not remain acceptable for large safety-critical programs, since they make behavior understanding and analysis almost impracticable. As in all other fields of computing, hardware independence will be forced by the fact that software has a much longer lifetime than hardware. Finally, it will be necessary and sometimes even required to formally verify the correctness of programs at least with respect to their crucial safety properties. All these new requirements call for rigorous concepts and programming tools and for the use of automatic verification systems.

The goal of this special issue is to present the *synchronous* approach to reactive and real-time systems, as well as the associated software tools and

verification techniques. The synchronous approach is based on a relatively small variety of concepts and methods based on deep, elegant, but simple mathematical principles. Roughly, the main idea is to first consider ideal systems that produce their outputs synchronously with their inputs. Such synchronous systems compose very well and turn out to be easier to describe and analyze that asynchronous ones. Furthermore, sophisticated algorithms can take advantage of the synchrony hypothesis to produce highly efficient code. Of course, the object codes are not really synchronous, but they are often of predictable behavior unlike fully asynchronous code (predictability is a key to correctly deal with speed issues of actual implementation; we shall not study this point here, referring to the specific papers). Automatic algorithms can adapt the resulting code to distributed architectures.

The synchronous programming concept was first introduced for software in [14, 15, 16, 17], but one must say that it bears many similarities with classical hardware concepts: in a clocked digital circuit, communication between subcomponents behaves as fully synchronous provided the clock is not too fast. Clock speed is predictable and all CAD tools can actually report to which clock speed a precise circuit can work.

Before presenting synchronous programming, we shall review the area of real-time systems and the presently prevalent programming tools

2 Reactive and Real-Time Systems: Examples and Main Issues

It is not our purpose to be exhaustive, but we feel it is necessary to analyse some examples of how diverse reactive and real-time systems can be. We shall first present the main application areas. We shall then present two case studies in more detail. Finally, we shall mention the main issues in reactive and real-time system development.

2.1 Application Areas

We list the applications areas in increasing order of complexity:

- **Pure task sequencers** are typically encountered in command boards, man-machine interfaces, or more generally Computer Integrated Manufacturing (CIM). They deal with *sequence of tasks* such as

```
PUT_OBJECT_ON_BELT; BELT_IN_MOTION;
```

DETECT_OBJECT; GRASP_OBJECT

Several elementary sequences may occur *in parallel* and cooperate for instance via shared events (PUT_OBJECT_ON_BELT can refer to events shared by a robot and by a belt). Task sequencers can be objects of high combinatorial complexity, so that the main issue here is to provide a formal method to convert a *specification*, i.e. a description that is easily understandable, into an efficient *implementation*, for instance the transition table of an automaton.

- **Communication protocols** are encountered in various kinds of networks, and in particular in real-time local-area networks. Similar comments may be drawn as for tasks sequencers, in particular as far as combinatorial complexity is concerned.

- **Low level signal processing** of which sensor data processing and signal processing in digital communication systems are typical instances. Digital filtering is here the basic item. At first sight, it can be considered as the direct adaptation of analog fitering to digital computing techniques. But the rapidly growing use of *adaptive* filtering makes digital signal processing evolve towards a computationally intensive real-time activity. The main issue is to achieve high throughput, so that it is desirable to handle both *algorithm* and *architecture* within the same framework.

- **Industrial process control** involves regulators that are supervised via internally or externally generated interruptions and sequential tasks. The main challenge is to provide a tool that is flexible enough to support an easy specification, and powerful enough to guarantee that the actual implementation meets the specification.

- **Complex signal processing systems** such as radar and sonar involve preprocessing of signals, followed by drastic data-compression via detection-and-labelling, and then by logically complex data-processing modules (data fusion, decision handling, ...). This results in computationally intensive real-time systems where many events are generated and further combined to fire new computations. The same remarks hold as for process control. The issue of speed becomes much more important.

- **Complex Control-and-Monitoring systems** govern aircraft and transportation systems as well as hazardous industrial plants. They can involve thousands of sensors, hundreds of actuators, and dozens of interconnected computer systems. Data can be processed in numerous operating modes, for example for maintenance or safety purposes. Heuristics of high combinatorial complexity typically may compose up to 90% of the application software code. Highly distributed target architectures must be considered. The safety constraints are obviously critical.

- Finally, C^3-systems (Command-Control-Communicate) or even C^3I-systems ('I' for "Intelligent") are encountered in military systems, in air traffic control systems, and also in large ground transportation systems. A further difficulty here is the highly distributed nature of the architecture supporting the real-time system: subsystems are moving, so that communication links cannot be considered as time-invariant.

2.2 A First Case Study: Automobile Control

Transportation systems involve numerous reactive systems, some of which bear severe real-time constraints. Les us take an automobile as an example.

There are or will be specific controllers for fuel injection, brakes, suspension, direction, etc. Each of those involves reactive programs that do numerical computations and have numerous functioning states, in particular because of hardware failure handling.

In the future, all these controllers will not stay independent of each other. They will have to be linked together for global coordination, for instance, to make cars lean inwards in curves. Coordination can be performed in two ways: either by distributing information from each controller to the other ones, thus making each of them much more complex, or by building a centralized controller that is itself a complex reactive system. In both approaches, there will be no easy solution and the safety problems will greatly increase. Linking the controllers together will be done by local area networks, involving themselves fast protocols which are non-trivial reactive programs.

At the user end, panels and man-machine interfaces will be computerized. Again, this will involve numerous reactive programs. Furthermore, one of the essential functions in car automation will be failure detection and reporting. This difficult area is often under-estimated: the messages to the

user or reparer should be simple and should not involve dozen of individual failures. This will require a clever mixture of reactive programming, signal processing, and heuristics.

Similar situations of course appear in almost all transportation systems. For automobiles, there is a rather strong additional constraint: the price of hardware should as small as possible, which means that programs are also subject to severe size constraints.

2.3 A Second Case Study: Speech Recognition Systems

Speech recognition systems are do not bear hard real-time constraints : the time response between the input (spoken language) and the output (text on screen or input to some other system) may be only loosely constrained. Nevertheless, the continuous speech signal must be processed on-line to avoid unbounded buffering. Hence continuous speech recognition is a good prototype of application where high-speed numerical preprocessing as well as complex symbolic postprocessing is required. Similar examples are found in data communication, pattern recognition, military systems, process monitoring and troubleshooting systems. We describe here briefly the speech-to-phoneme recognition system developed at IRISA [7]. Its overall organization is shown in the figure 1. The originality of this system lies in its use of a *segmentation* of the continuous speech signal prior to any recognition. The *automaton* supervises the segmentation; it fires small modules to compute *cepstra*, a representation of the spectral characteristics of the signal, associated with detected segments as well as some *acoustic/phonetic cues*. All these modules are numerically oriented. Finally, high level processing is performed following a technique close to *Hidden-Markov Model* (HMM) methods [22] : maximum likelihood decoding based on a stochastic automaton. This is again a numerically as well as logically oriented module.

To illustrate further how signal processing algorithms may give rise to reactive systems, let us give additional details on the segmentation module. The outcome of this processing is shown in the figure 2. The segmentation procedure is mainly numerically oriented and is performed on-line. Detection of change occurs with a bounded delay, so that the speech signal must be reprocessed from the estimated change time. Furthermore, some local backward processing of the speech signal is also needed. Hence, while this is still a real-time processing of speech signal, its timing is far from being trivial. Therefore, writing a real-time oriented programming of this processing in C or FORTRAN is a tedious and error-prone task.

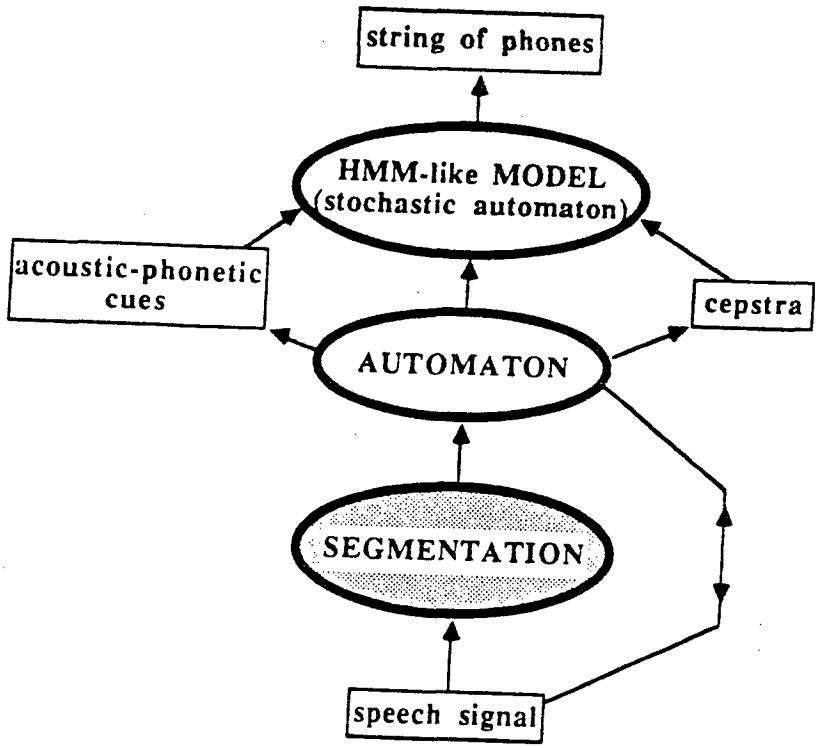


Figure 1: A speech-to-phoneme recognition system

To summarize, this example is a good prototype of a complex real-time signal processing application. It may be compared to radar systems for example.

2.4 Reactive and Real-time Systems — Major Issues

Most reactive and real-time systems naturally decompose into communicating concurrent components. The programming architecture must follow this decomposition. Hence, all aspects related to concurrency are important: communication, synchronisation, organisation of the computational flow. We shall refer to these aspects as qualitative ones. The timing constraints imposed on real-time systems also impose to consider quantitative aspects mostly related to the speed of computations. Here are the major issues related to these aspects:

- **Use modular and formal techniques to specify, implement, and verify programs.** The specification-inplementation cycle is a major issue in the software life cycle. Modular programming is necessary to reflect the conceptual architecture into the programs themselves. Relying on a discipline of programming based on manual trans-

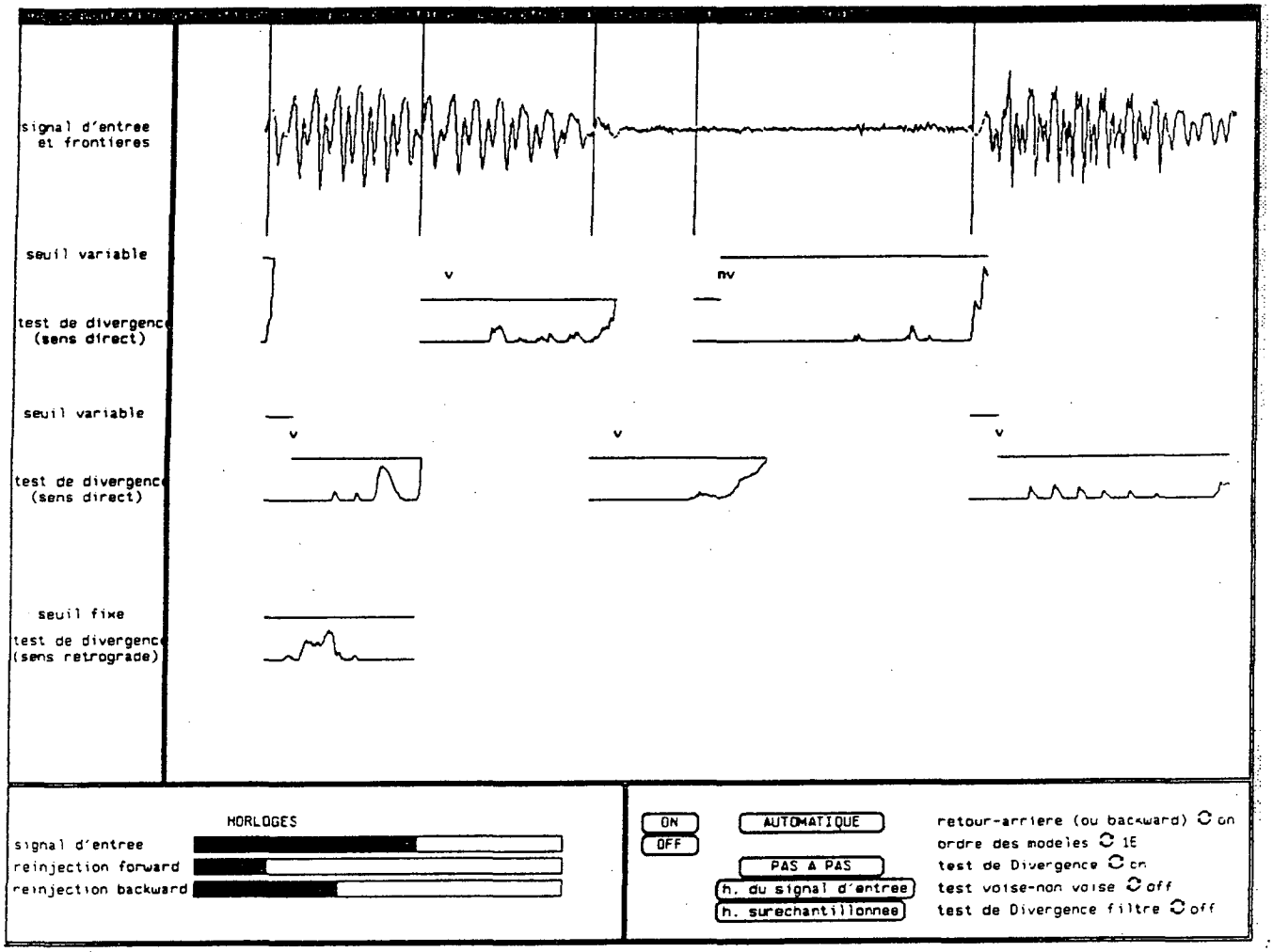


Figure 2: The segmentation module. The detected segments are superimposed on the signal (top line). Subsequent lines show the behaviour of several auxiliary quantities (the divergence tests) that are computed on-line to perform the segmentation. As a byproduct of the processing, the auxiliary labels "v" and "nv" indicate voiced and unvoiced segments respectively.

lations from specification to implementation is known not to guarantee enough safety. One should therefore provide modular tools that formally and inherently guarantee the equivalence preserving throughout the specification → implementation process and give access to formal verification techniques. Notice that these tools must perform non-trivial transformations, since there is usually no perfect match between the functional architecture and the target computer architecture.

- **Encompass within a single framework all reactive aspects,** i.e. communication, synchronisation, logic, computational flow. Having to deal with several frameworks can break the coherence of the global chain. This constraint may be somewhat relaxed if the interface between frameworks is very cleanly defined and permits useful reasoning.

- **Deal with distributed target architectures.** The need for distributed architecture can come either from performance requirements or from geographical constraints within the application.

- **Preserve determinism whenever possible.** A system is said to be deterministic if a given sequence of inputs always produces the same sequence of outputs. Any sensible functional description of the kind of real-time system we discussed (information processing, control, C^3, ...) should be obviously deterministic in this sense : there is no reason the engineer should *want* its procedure to behave in some unpredictable manner. Furthermore, even when the implementation of a complex system is globally non-deterministic, most parts of it are individually deterministic. Since deterministic programs are much simple to analyze and debug that non-deterministic ones, tools should not force non-determinism unless specifically required to. There are obviously subtle trade-offs between determinism and concurrency when implementation issues are considered.

- **Consider issues of speed.** In all cases, the executable codes should be efficient and avoid overheads due to unnecessary run-time communications. Execution times should be predictable whenever possible. For real-time systems, if object code efficiency is not enough to guarantee the respect of timing constraints, timing issues should be incorporated in the model.

3 Real-Time Programming: the State of the Art

We review the techniques classically used for real-time according to the above issues (see [9] for a more complete presentation):

- *Connecting classical programs by making them communicate using OS primitives.* This is the most common way of doing things. There is presently a lot of experience of using this technique, but its drawbacks are rather numerous and severe. There is no *single* object to study, but a set of more or less loosely connected programs. Understanding, debugging, and maintaining applications is hard. For the same reason, there is little room for clean automatic program behavior analysis, and therefore no way of formally guaranteeing safety properties. Last, operating systems are generally somewhat non-deterministic, unless they are reduced to trivial sequencers, which in turn makes programming harder.

- *Using finite-states machine*, also called *finite automata.* These objects have numerous advantages: they are deterministic, efficient, they can be automatically analyzed by numerous available verification systems. However, they have a severe drawback: they do not directly support hierarchical design and concurrency. A small change to a specification can provoke a complete transformation of an automaton. When they are put into cooperation, separately small and pretty automata can yield a big ugly one. As soon as they are large, automata become impossible to understand for human beings.

- Using *Petri Nets* or Petri-Net based formalisms such as the GRAFCET [24, 11]. Such formalisms are commonly used for comparatively small applications. They naturally support concurrency, but they lack modular structure and often lack determinism. They do not scale up well to big applications.

- Using classical *Concurrent Programming Languages* such as ADA [6] or OCCAM [13]. These languages take concurrency as a primary concern and support modularity. They permit their user to see a single program for a concurrent application. However, they are essentially asynchronous and non-deterministic: although a communication is seen as a synchronization between two processes, the time taken between the *possibility* of a communication and its actual *achievement* can be arbitrary and is unpredictable. When several communications can take

place, their actual order is also unpredictable. For all these reasons, such languages are hardly adequate for real-time programming. Finally, automatic program verification is often not feasible since asynchrony makes the programs state spaces explode. See [9] for more details.

4 The Synchronous Approach to Reactive and Real-Time Systems Specification, Design, and Implementation.

We now turn to the synchronous approach and show that it reconciles all aspects discussed in the previous sections: it makes deterministic hierarchical concurrent specification and programming possible, it leads to efficient and controllable object code, and it makes it possible to use automatic verification tools by avoiding or at least reducing the state space explosion problem.

The basic idea is very simple: we consider *ideal* reactive systems that produce their output *synchronously* with their input, their reaction taking no observable time. This is akin to the instantaneous interaction hypothesis of Newtonian mechanics or standard electricity, an hypothesis which is well-known to make life simple and to be valid in most practical cases. The main simplification lies in the fact that sets of ideal system compose very well into other ideal systems. In the synchronous model, a system can be decomposed into concurrent subcomponents at will without affecting its observable behavior even w.r.t. timing issues.

To illustrate the synchrony hypothesis, we shall start from two extreme examples. First, we discuss the case of sequential tasks. Then, we discuss the case of regulators in process control or adaptive filtering in signal processing. Based on the first example we introduce the synchronous model as an idealization of reactive systems where internal actions and communications are instantaneous. Based on the second example, we introduce the synchronous model as dealing with system of interconnected dynamical equations (the block-diagrams of signal processing or control sciences), or, equivalently, as a description of the traces. Then we show how both points of view may be interchanged or mixed together, leading to an idealized picture of *general* real-time systems.

Note that it ist not our purpose to be formal in this introductory paper. We simply present an intuitive picture of the synchronous style of mod-

elling we want to promote. Information on related formal models and their properties can be found in the subsequent papers and references therein.

4.1 A First Example: Clicking on a Mouse

We consider a mouse handler that has two inputs:

- CLICK : a push-button,
- TICK : a clock signal.

A first CLICK fires the GO module that watches for the elapsed time to decide whether a SINGLE, or a DOUBLE CLICK has been received (on the diagram of the figure 3 the maximum elapsed time is 4). The end of the enabling period where the CLICKs are watched for is indicated by the signal RELAX.

Obvious modularity considerations lead to consider this small system as the composition of two communicating subsystems, namely:

- a module GO that is fired by the first click and delivers RELAX at the end of the enabling period,
- a module SIMPLE_MOUSE that outputs signals SINGLE, or DOUBLE according to the above specification when it receives RELAX.

Both modules and their resulting communication we call MOUSE are shown in the modular state transition diagram of the figure 3. This figure shoud be read as follows. Each of the two modules contains a state transition diagram ; transitions are labelled with words that list the events which must occur simultaneously with the considered transition. When two different words are assigned to a transition, then any one of them may cause the transition to occur. The two modules share RELAX as a common event, which means that each time one module executes a transition involving RELAX, then the other one must execute simultaneously some transition involving the same event. This presentation roughly follows the STATECHARTS style [3]. Such a specification of this toy system can be intuitively accepted by the reader. This diagram, however, should be interpreted according to the following rules :

1. Changes of state in each of the modules should be considered as *synchronous* (or simultaneous) with the reception of the mentioned input signals.

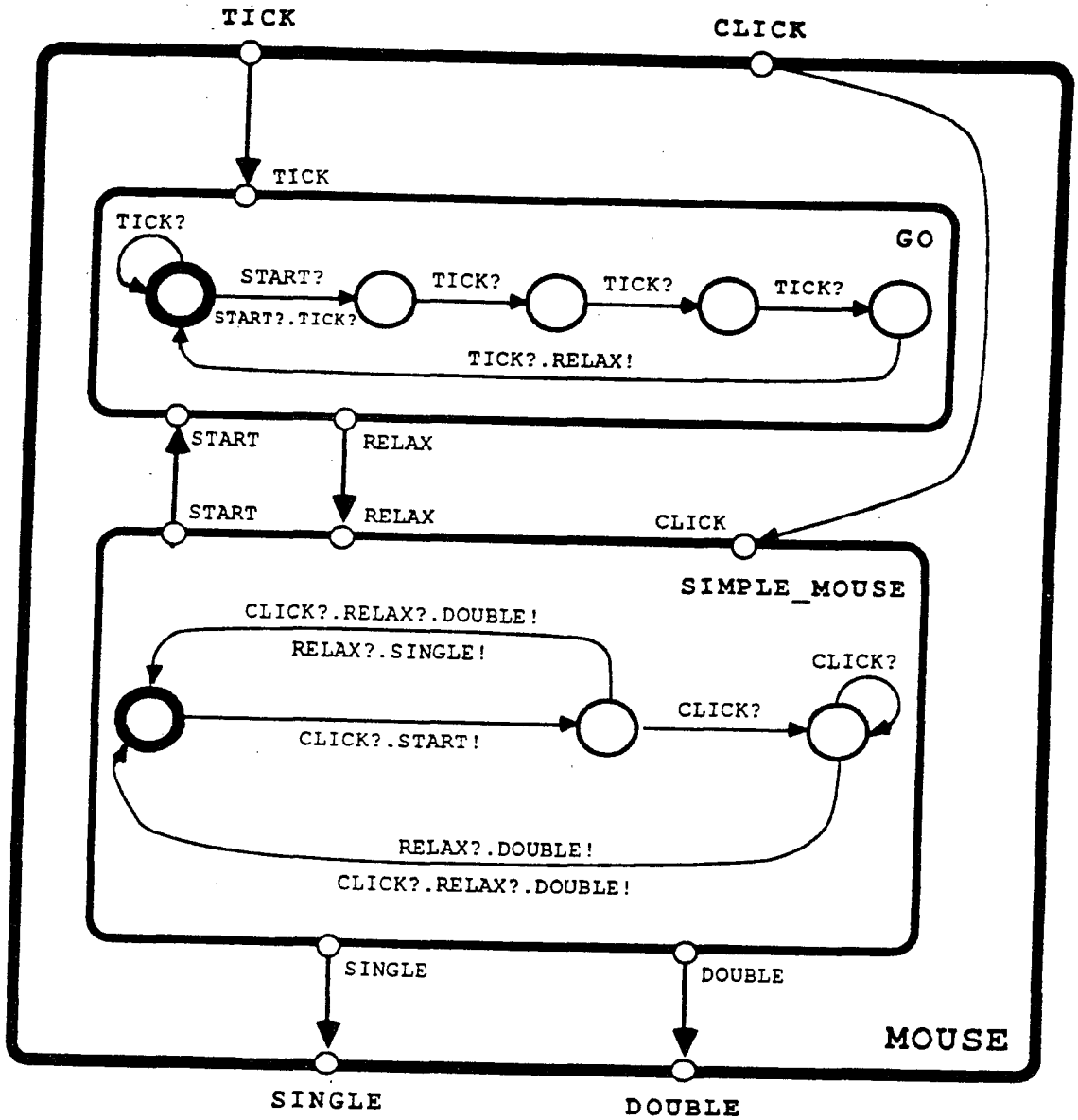


Figure 3: The modules GO, SIMPLE_MOUSE, and their communication MOUSE.

2. The emission of output signals in each of the modules should be considered as *synchronous* (or simultaneous) with the associated change of state.

3. The communications follow the principle of "instantaneous broadcast" of the signals emitted by the modules, which means that their reception is *synchronous* (or simultaneous) with their emission.

4. The output behaviour of MOUSE is entirely fixed whenever the *global interleaving* of the two input signals TICK, CLICK is given by the environment.

To summarize the three first points, *internal actions and communications are instantaneous.* As a byproduct, outputs are synchronous with inputs as requested above. The fourth point follows and implies that determinism is preserved by synchronous concurrent composition.

An example of a global interleaving is given in the following chronogram, where signals written on the same column are simultaneous and events are ordered from left to right:

TICK	TICK CLICK	TICK	TICK	 CLICK	TICK	TICK

This chronogram must be understood as a *discrete event* one. Only the global ordering makes sense, the interval between successive events does not need to be constant with respect to some externally given notion of absolute time. Actually, no physical notion of time is referred to in the mouse specification, although in practice the TICK input will often be generated by actual quartz clocks.

The synchronous model does *not* specify how an input chronogram is generated by the environment. This relies on the actual implementation of the mouse as an electronic device, using simple sensors and A/D converters. Then providing a global input interleaving can depend on some comparison of the actual instants of arrival of physical signals actually bound to continuous time,

The mouse example reveals a fundamental feature of our approach to real-time programming: thanks to the above idealization of synchrony, the reactive part of our system is made *implementation independent*, and only a relatively very small part –building the global interleaving– is implementation dependent and bound to physical time. Most programming difficulties actually arise in the reactive part in actual reactive problems. Later

on, we shall see that powerful formal reasoning can be performed on the implementation-independent part, while some formal reasoning can be still also performed on the implementation-dependent depending on the cases.[2]

We must recognize that we left aside the issue of *computation speed* in this discussion, since we assumed an infinitely fast machine was at hand. But it turns out that this is too dogmatic an interpretation of the synchronous model and that we can be more flexible. As an illustration, consider again the chronogram above augmented with the outputs corresponding to each input:

TICK	TICK CLICK	TICK	TICK		TICK	TICK
	START			CLICK		
						RELAX DOUBLE

Now, assume the mouse system has been actually implemented in some environment subject to physical continuous "real" time, and that the real time unit is plotted on the horizontal axis. A realistic picture is to consider that the separating vertical lines are *elastic* ones, i.e. that they may be redrawn as oblique curves, provided that causality be preserved (outputs must follow inputs). This is exactly what is done when considering clock cycles in digital circuits. In some cases, we can even make slots overlap to perform pipelining. Taking into account physical time consumption at the implementation level amounts to reason about such a flexibility.

Altogether,we hope to have convinced the reader that there are two distinct issues: implementation-independent logical synchronization and qualitative timing on the one hand and implementation-dependent physical time consumption on the other hand. We further discuss this point in the section 5. It should be remembered that our synchronous model deals only with qualitative timing and synchronization and not physical time consumption. Our claim is that one should stay within the ideal synchronous model as much as possible and consider actual timing dependencies only when needed and where needed.

[2]for instance, we may prove here that any global interleaving is a possible input to this system.

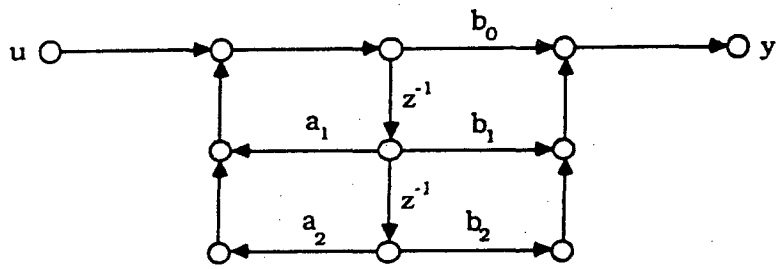


Figure 4: A second order filter

4.2 A Second Example: Digital Filtering

The signal flow graph of a "second order digital filter" in the classical direct form [21] is shown in the figure 4. At the nodes of this graph, incoming signals are added and their result is broadcast along the outgoing branches. The labels z^{-1} and a_i, b_j on the arcs denote a shift register and a multiplication by the mentioned constant gain respectively. Accordingly, the signal flow graph of the figure 4 is a coding of the following formula:

$$\begin{aligned} w_n &= u_n + a_1 w_{n-1} + a_2 w_{n-2} \\ y_n &= b_0 w_n + b_1 w_{n-1} + b_2 w_{n-2} \end{aligned}$$

where n denotes the time index. A little algebra yields equivalently :

$$y_n = a_1 y_{n-1} + a_2 y_{n-2} + b_0 u_n + b_1 u_{n-1} + b_2 u_{n-2}$$

We can read this mathematical expression as describing a machine which performs the specified filtering *according to the principles 1, 2, and 3 of synchronicity* we have introduced while discussing the mouse. This is certainly a well-accepted idealization of a digital filter.

A slightly more subtle example is the signal flow graph of figure 5, which represents a two-port filter derived from the preceding one. It corresponds to the formula

$$y_n = a_1 y_{n-1} + a_2 y_{n-2} + b_0 u_n + b_1 u_{n-1} + b_2 u_{n-2} + v_n \qquad (1)$$

where v is a second input signal. Two input ports are needed at the interface, as in the MOUSE example, and principle 4 applies here. But, what is new here is that *not every global interleaving is allowed* for the two input signals u, v: to each sample of u must correspond a unique sample of v.

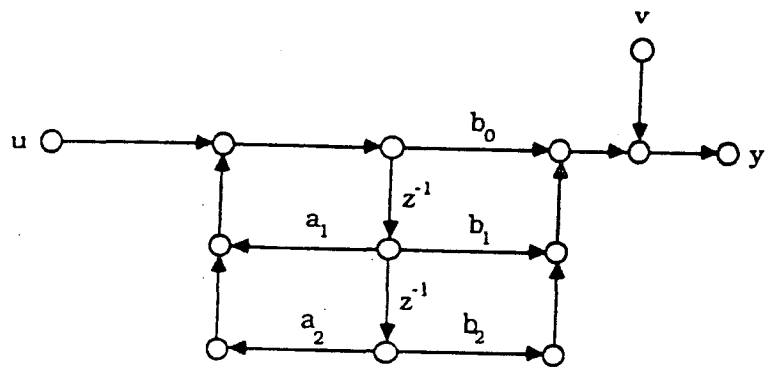


Figure 5: A two-port filter

Now, interconnecting digital filters is usually specified by linking graphs, or equivalently by writing systems of equations. For instance, the filter of the figure 5 may be redrawn in a modular way as shown in the figure 6. But this corresponds to replacing the (dynamical or recurrent) equation (1) by a *system* of equations in the usual mathematical sense :

$$\begin{aligned} z_n &= a_1 y_{n-1} + a_2 y_{n-2} + b_0 u_n + b_1 u_{n-1} + b_2 u_{n-2} \\ y_n &= z_n + v_n \end{aligned}$$

where common names denote the same signal.

4.3 Towards the Synchronous Modelling Approach

The mouse example was naturally described using state transition diagrams. The digital filter example was naturally described in the mathematical framework of systems of recurrent equations. Formal models corresponding to these different frameworks can be shown equivalent. We find it illustrative to perform the following exercise on these two examples : crisscross the models, i.e. describe the digital filter via a state transition diagram and the mouse via a system of recurrent equations.

A state transition diagram for the digital filter. To simplify our presentation, we shall replace the filter (1) by the simpler one

$$y_n = a_1 y_{n-1} + a_2 y_{n-2} + u_n \qquad (2)$$

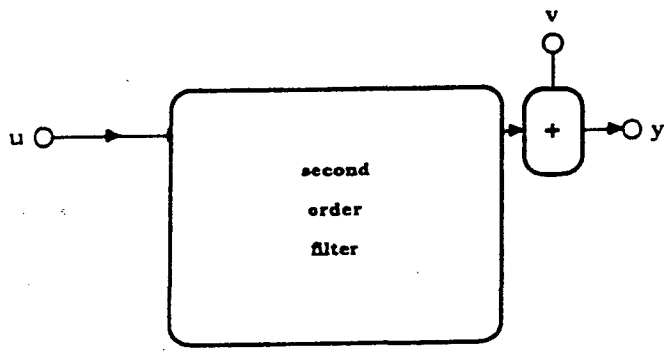


Figure 6: A two port filter: modular specification

Introduce the vector signal

$$X_n = \begin{bmatrix} y_n \\ y_{n-1} \end{bmatrix}$$

and rewrite (2) in the "state space" form

$$\begin{aligned} X_n &= \begin{bmatrix} a_1 & a_2 \\ 1 & 0 \end{bmatrix} X_{n-1} + \begin{bmatrix} u_n \\ 0 \end{bmatrix} \\ y_n &= \begin{bmatrix} 1 & 0 \end{bmatrix} X_n \end{aligned} \tag{3}$$

One time step of the system (3) would be written as follows in a standard sequential programming language :

$$X := \begin{bmatrix} a_1 & a_2 \\ 1 & 0 \end{bmatrix} X + \begin{bmatrix} u \\ 0 \end{bmatrix}; \tag{4}$$

$$y := \begin{bmatrix} 1 & 0 \end{bmatrix} X \tag{5}$$

This program is of the form (4);(5), i.e. it is composed of two instructions separated by the PASCAL-like sequencer ";". Denote by α the action performed by this program. Then the state transition diagram corresponding to the system (3) is shown in the figure 7. In this diagram, the state just counts the occurrences of the input signal. The task is in fact entirely summarized by the α label of the action (4);(5) which corresponds to a single iteration of the recurrent equation.

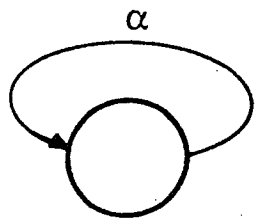


Figure 7: State transition diagram of the filter

A system of recurrent equations for the mouse. To simplify our discussion, we shall only consider the SIMPLE_MOUSE. We shall term an *event* the occurrence of at least one of the input signals CLICK and RELAX. Events will be indexed using the integers $\mathbf{N} = \{1, 2, 3, \ldots\}$. Then, we denote the subsequences of events where CLICK and RELAX are respectively received by $\mathbf{C} = \{C_1, C_2, \ldots, C_m, \ldots\}$ and $\mathbf{R} = \{R_1, R_2, \ldots, R_k, \ldots\}$. This simple mouse can be specified by the following system of equations, where the running index n denotes the current event:

$$\mathbf{N} = \mathbf{C} \cup \mathbf{R} \tag{6}$$

$$X_n = \begin{array}{l} \text{if } n \in \mathbf{R}, \text{ then } 0 \\ \text{else } \min\{2, X_{n-1} + 1\} \end{array} \tag{7}$$

$$M_{R_k} = \begin{array}{l} \text{if } R_k \in \mathbf{C}, \text{ then } \min\{2, X_{R_k - 1} + 1\} \\ \text{else } X_{R_k - 1} \end{array} \tag{8}$$

$$\text{if } R_k \in \mathbf{C}, \text{ then } X_{R_k - 1} \neq 0 \tag{9}$$

- Equation (6) specifies that events consist of the occurrence of at least one of the inputs CLICK, RELAX.
- X denotes the internal state of the counter. Equation (7) expresses that X is reset to 0 whenever RELAX is received and incremented whenever CLICK is received but not RELAX. Note that the specification (6) is used for (7) to be correct: since, according to (6), the index n of events is incremented only if at least one input signal has been delivered, the "else" in equation (7) means $n \notin \mathbf{R}$ and thus $n \in \mathbf{C}$.
- The integer M_{R_k}, whose possible values are $1, 2$ is the output. Equation (8) specifies that the output M has the same index as R; the value

carried by M is either the previous value of the state (when RELAX is received alone), or the previous value of the state incremented by one (when both CLICK, RELAX are received).

- Finally, equation (9) asserts that RELAX cannot occur when the counter is in its initial state 0.

Hence we should call this a *Multiple Clocked Recurrent System*, since different time indices are used here. Obviously, handling more than 3 different indices in such a pedestrian way becomes untractable. The model (6,7,8,9) also reveals clearly that the two subsequences $C_1, C_2, ...$ and $R_1, R_2, ...$ are used. But knowing these consists precisely in knowing the global interleaving of the two input signals : this is precisely point 4 of the principles of synchronicity. Again, no physical notion of time is used here. Finally the model (6,7,8,9) consists of describing *relations* between various signals rather than constructing a machine whose behaviour represents that of the desired mouse. In particular, equation (9) specifies a *constraint* on the input signal RELAX.

4.4 Summary of the Synchronous Model

The discussion above illustrates that two different in style but equivalent forms of *synchronous modelling* may be used. Both specify an ideal real-time machine with the following features:

1. *output is synchronous with input, internal actions are instantaneous, communications are performed via instantaneous broadcasting,*

2. *the global interleaving of the external communications may be partially chosen by the environment and is essential in analysing the behaviour of the system.*

The two styles are:

State based formalisms. In the mouse example, we used state transition diagrams where arrows were labelled by communication actions. The Statecharts generalize this kind of presentation. The CSML and ESTEREL formalisms have a fairly similar but more implicit notion of state based on control positions in an imperative program. All these formalisms will be presented in this special issue. The corresponding formal models are discussed in the papers above or in the references therein.

Multiple Clocked Recurrent Systems (MCRS). They are a way to describe the legal traces of a system and are generalizations of the usual models of dynamical systems used in digital signal processing or control. This generalization is needed to handle different timings and their relations, which naturally arise in complex real-time applications. The languages LUSTRE and SIGNAL, [4, 5] presented in this special issue section mainly rely on this style of modelling; proper references to corresponding formal models can be found in these articles.

State-based formalisms are easy and natural to use in problems where control flow is prevalent, for example for systems that often jump between many distincts functioning modes (man-machine interfaces, protocols, control panels, etc.). Writing concurrent components is easy at a syntactic level, but defining the behaviour of a concurrent composition is not easy: broadcasting signals has the effect that concurrent components constrain each other in a non-trivial way at each reaction. The overall behavior is given by a fixpoint of a set of constraints, generally computed using formal semantics given in Plotkin's *Structural Operational Semantics* inference-rules based style. Roughly speaking, SOS are the convenient framework to handle state-based formalisms in a modular style, just as if they were systems of equations.

MRCS are clearly well-adapted to problems where data flow is prevalent, signal processing being an obvious example. The composition of MRCS is very easy to define since they are standard mathematical equation systems. Conversely, MRCS are weak where state-based approaches are strong, that is when the complexity is in functioning mode changes. Then the user must handle explicit control variables to record the current mode, not an easy task.

It is shown in [8, 12] that both styles allow to describe the reactive aspects of all real-time system. In practice, each style tends to be weak where the other one is strong. Since we do not know yet how to combine both styles in a common formalism nor whether this makes sense, we need to use both in real applications, depending on the style of individual parts. There is some present work not reported in this special issue to make both styles as compatible as possible, for example at the object code level.

4.5 Solving Communication Equations

Be it in the state-based approach or in the MCRS approach, communication equations may have:

- **no solution** : the constraints contradict each other, or cycles of causality may exist that cannot be solved using finite algorithms. Such contradictions or deadlocks may involve the whole system, or only a subsystem of it.
- **infinitely many solutions** : the timing of the various signals is not completely determined by the given inputs, we get nondeterminism,
- **a single solution** which is also an input-output map : our program is deterministic, and is thus a suitable candidate for proper execution.

All languages presented in this special issue have specific algorithms to check these properties. In particular *determinism can be checked and guaranteed,* an important feature as we have discussed before.

4.6 Program Verification

In most reactive or real-time applications, it is important to be able to formally verify program properties: liveness os safety properties, respect of total or partial specifications. There are various available software tools to perform such verifications for the formalisms described in this special issue. Some use model checkers to compare the infinite sequence of events of a given program with a list of specified properties that are stated using a different formalism, see for instance [1, 3] where temporal logic is used for this purpose, and also [2]. Some other tools provide the user with abstractions of the program, i.e. with reduced programs that behave as the original one but involve only a (small) subset of signals, see [2, 4] for such an approach. Finally, in MCRS formalisms such as SIGNAL [5] and LUSTRE [4], constraints or properties can be specified just as further dynamical equations that must be implied by the given system. Then there is no deep distinction between program and safety properties and the standard program compilers can act as verifiers.

5 Synchronous Models Versus Asynchronous Systems

Actual machines for which the ideal synchronous model is realistic do exist. For instance, strongly synchronized hardware or VLSI architectures are such that internal actions and communications occur within a clock cycle, that

is within a "tick" in our sense. The only difference is that outputs are given to the environment at the end of the cycle and not synchronously with the inputs. Since the cycle time is very short, say 100 nanoseconds, this is the best approximation we can get. The language CSML [1] or the hardware implementation of ESTEREL and LUSTRE [10] implement this point of view.

However, most of the machines used to support the applications we listed in the section 1 should be certainly considered as *asynchronous* in any reasonable sense. Furthermore, real-time systems are often implemented on distributed architecture, that is on sets of processors connected by asynchronous means. Synchronous models as introduced before can hardly be considered as realistic for such target architectures.

In this section, we discuss implementation issues when asynchronism must be considered. We first consider the case of the digital filter and exhibit different *realistic* implementations for which we can prove equivalence with the original specification. Then, we consider a simple example of token-based architecture as an instance of asynchronous machine and show how reasoning on its synchronisation may be performed via considering an associated synchronous model.

5.1 Implementing the Digital Filter

An infinitely fast machine implementing equation (3) is certainly a correct implementation of the digital filter of figure 7, but it is obviously an unrealistic one. We shall discuss two relevant alternatives.

A purely sequential implementation can be derived from the signal flow graph of the figure 5 in the following classical way. First consider the associated *dependency graph* obtained by cutting the branches labelled with a delay z^{-1} as shown in the figure 8. We get an acyclic directed graph. Peeling this graph by removing first the input nodes and then subsequent ones yields a sequential execution scheme of each single time step of the system. This is depicted in figure 9.

A data–flow (asynchronous) execution can be simply derived by interpreting each node and branch in the graph of the figure 5 according to the data–flow mechanism shown in the figure 10. What is important here is that *we know before execution that this token mechanism will be nonblocking and with bounded files.* This property is well-known; it is already used to guarantee well-behaved executions for simple data–flow machines, see [19].

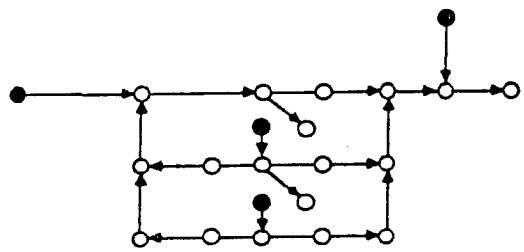


Figure 8: The dependency graph corresponding to figure 5

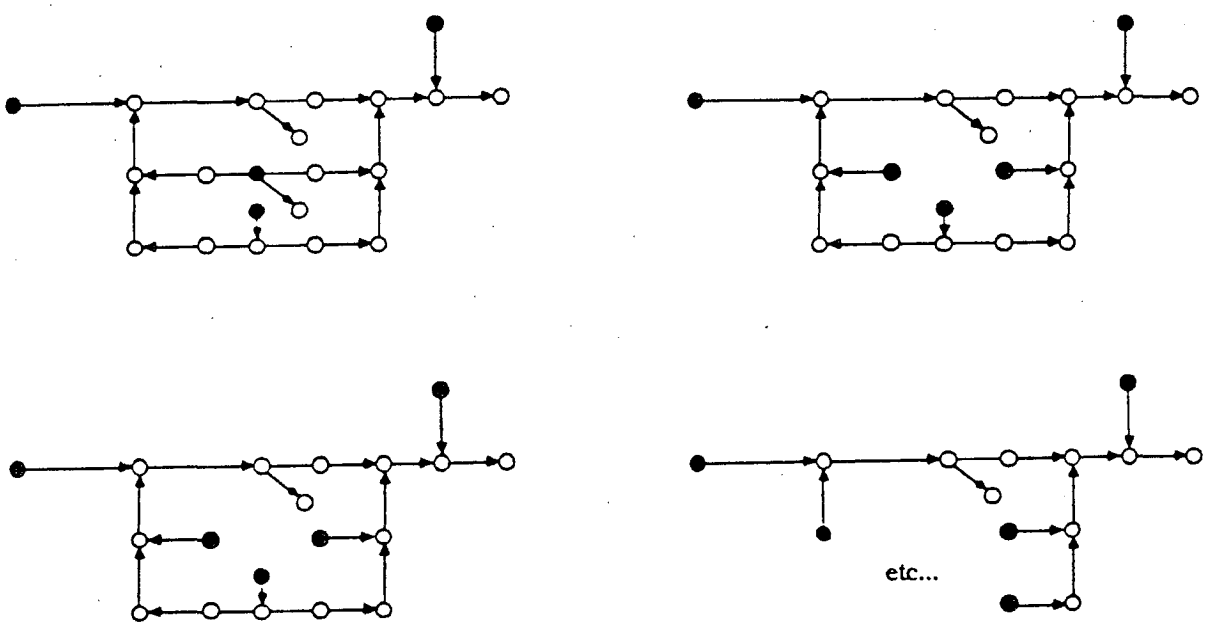


Figure 9: Peeling the graph of the figure 8.

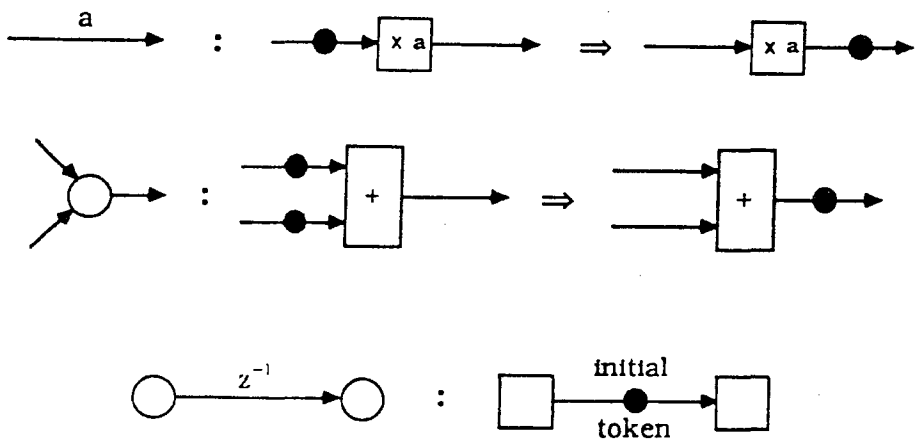


Figure 10: Data-flow mechanisms for the graph of the figure 5

Note that similar arguments can be used to justify asynchronous executions à la Petri net of this filter.

This ability to validate asynchronous executions of our synchronous ideal machines generalizes to the fully general reactive systems we can model with our approach. It is beyond the scope of this paper to formally justify this claim in a general fashion. We just present a simple example and show how to associate a synchronous model with a "generalized" data-flow machine [20] to validate it.

5.2 Validating Asynchronous Machines with Synchronous Models

The figure 11 depicts the data-flow actors introduced in [20].[3] Let us concentrate on the SELECT operator, and consider the run depicted in the figure 12. We construct a global "time indexing" of the tokens which is be consistent in the following sense: *the tokens that are consumed or produced in a given firing must have the same time index.* By inspecting the run of the

[3] they were in fact inspired by the primitive operators of the SIGNAL synchronous language we present in this special issue.

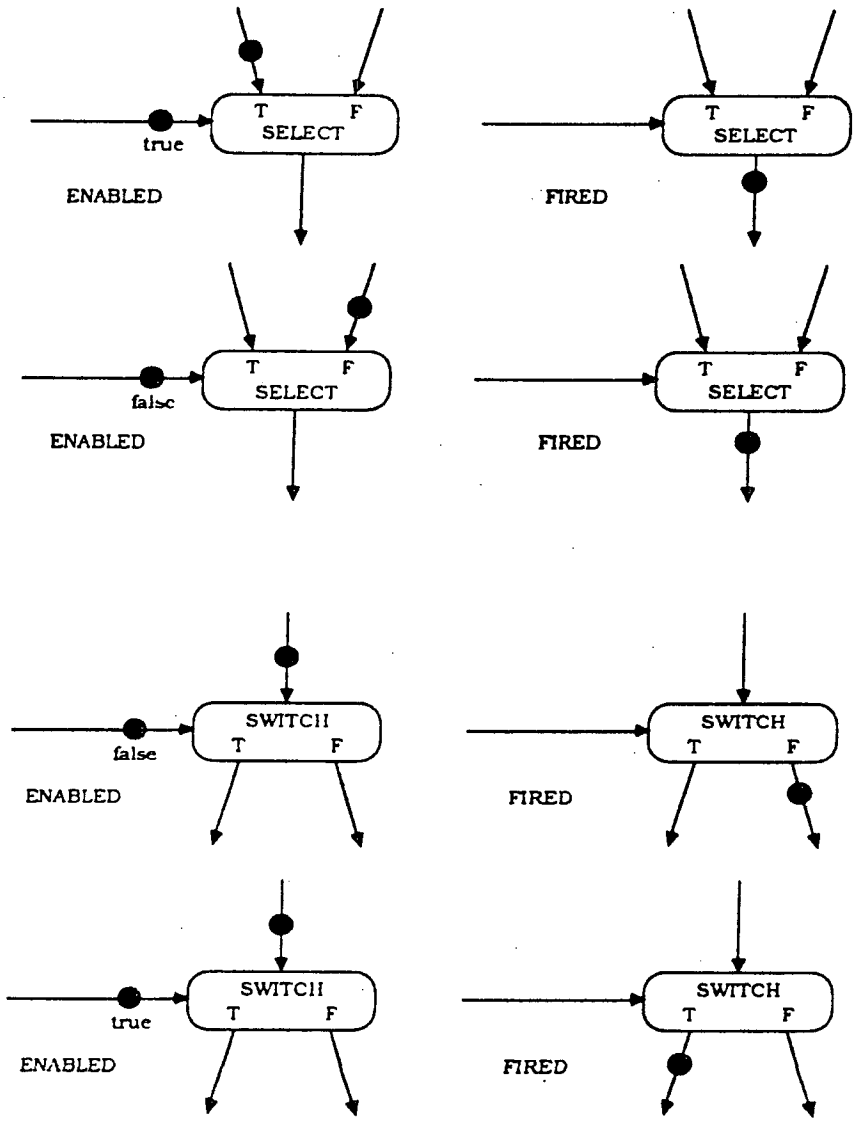


Figure 11: The data-flow actors introduced in [20].

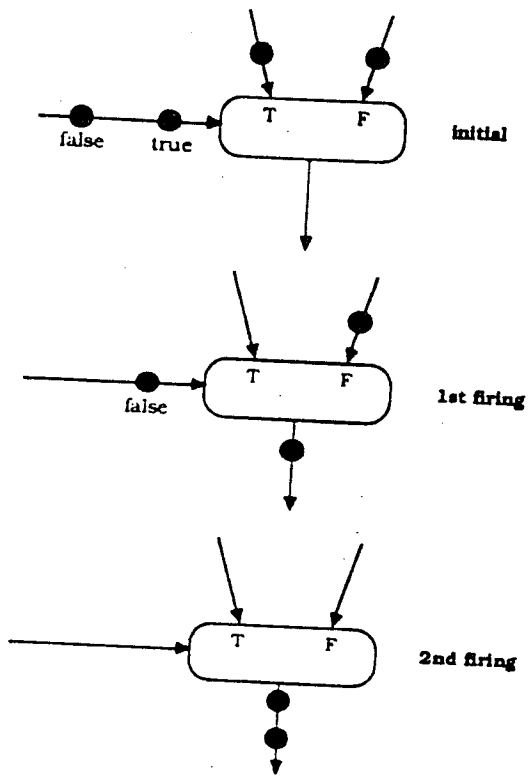
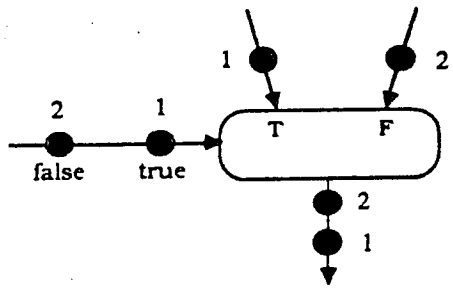


Figure 12: A run of SELECT



or, better:

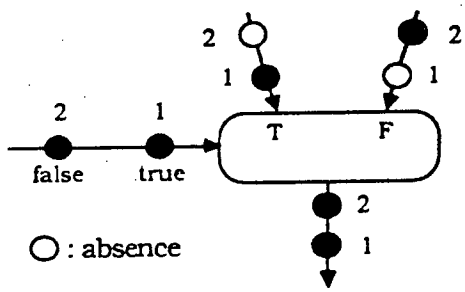


Figure 13: A consistent time indexing of the tokens.

figure 12, one easily checks that the time indexing shown in the figure 13 is consistent in the above sense. Let us collect the tokens with the same label into successive slots. We get a global interleaving of the four signals involved in this actor as shown below:

T input:	T_1		...
F input:		F_2	...
boolean:	*true*	*false*	...
output:	T_1	F_2	...

What we hane derived here is a synchronous model associated with the data–flow actor. Generally speaking, given a data–flow graph built with the above primitive actors, we can automatically build a synchronous model as an interconnection of synchronous subsystems associated with each actor. Then any of the formal verification methods presented in this special issue can be applied to the obtained synchronous model. It turns out that correctness of this synchronous model[4] guarantees a satisfactory execution of the original data–flow graph for any input data sequence.

5.3 The Synchronous Approach to Asynchronous Implementations

Let us summarize the preceding discussions by the following facts:

1. when feasible, strictly synchronous executions of synchronous systems are certainly valid (cf. VLSI and hardware),

2. verification and proofs of correct synchronisation and logic are available in the synchronous approach to real-time programming,

[4] cf. the remark at the very end of the section 4.

3. a sequential execution scheme can be derived at compile time for any synchronous system,

4. the idealized strict synchronicity hypothesis can be relaxed to yield fully *asynchronous* executions of synchronous systems that are guaranteed correct,

5. the formal verification tools based on the synchronous approach provide a way to validate asynchronous executions.

Since both purely sequential (eg. Von Neumann) and purely asynchronous execution schemes cam be associated with synchronous systems, it is easy to believe that *mixed sequential/asynchronous* execution schemes can be derived as well. To conclude, using the synchronous and asynchronous frameworks in the above suggested way yields a much cleaner treatment of the specification → implementation process. Again, we should point out that issues of physical time consumption are not considered here as such; however, we think that our approach facilitates their proper handling. This special issue reports various experiments along this line.

6 Possible Impact of the Synchronous Approach on the Activity of Real-Time Programming

The techniques we presented here are clearly novel. This has some consequences we discuss now.

The synchronous formalisms are based on very advanced and powerful concepts and have clean mathematical semantics. This is clearly a big progress compared to previous tools. However, two questions are still largely open: that of user interfaces and that of programming methodology.

Consider first user-interfaces. Some formalisms are purely graphical (Statecharts), some are purely textual (CSML, ESTEREL), and others can use both graphical and textual presentations indifferently (SIGNAL, LUSTRE). Speaking first of graphical interfaces, STATECHARTS are state-oriented while the the block-diagram interface of SIGNAL is data-flow oriented. None of these two choices covers the whole area of reactive and real-time systems: state-oriented diagrams are poor for signal processing and block-diagrams are poor for state machines. When using textual formalisms, one often needs to draw pictures to explain program architectures, but there is yet

no clear way to make these drawings formal rather than simply explanatory. Therefore, while the principles of the synchronous approach have a wide applicability, this is hardly the case for the particular user interfaces available so far. The development of rich and well-targeted user interfaces for synchronous languages must be a techical priority.

Les us now turn to methodology. At least in the area of real-time systems area, most potential users have a process-oriented background[5] rather than a computer science oriented one. Furthermore, most of them are used to a particular way of thinking, say for example to state-based reasoning rather than to equation manipulation. Since the synchronous approach yields new design and programming styles, one should develop methodologies that make these styles easy to master. Such methodologies do not really exist yet and their development will take some time. They should of course be based on elaborate software development environments and on fancy user-interfaces.

Tools that are considered as user-friendly in a particular application domain do exist: we can cite for example the GRAFCET. However, their associated formalisms definitely lack precise semantics. While this can be accepted in simple situations, it becomes unacceptable when safety is critical. There might actually be a reasonable way to make a smooth transition from existing tools to really rigorous ones: to build programming environments externally based on existing formalisms but internally based on the synchronous approach and on rigorous semantics.

Finally, it is important to note that synchronous languages are not completely bound to nondeterminism. Some of the synchronous languages perfectly well accept nondeterministic programs as modules, although they refuse to produce deterministic code out of them. Nondeterministic modules can be useful to model the environment or the controlled physical process . This might be the basis for a design methodology of real-time software based on a joint handling of the application and of a model of the physical process. Such an approach is standard in control systems design; it is interesting to note that it might become valid for real-time programming as well.

[5] they are typically chemical, mechanical, aircraft, control engineers, etc...

7 Conclusion

We have first discussed the major issues in the area of reactive and real-time programming, insisting particularly on safety constraints. We have then informally presented the new synchronous programming approach. Based on simple examples, we have discussed two orthogonal synchronous styles and their semantics: a state-based style and a data-flow based style. Each style applies to a particular class of problems; complex applications will certainly require the cooperation of both. We have briefly discussed how to verify program properties and how to make asynchronous implementations look like synchronous ones.

The other articles of this special issues will present the existing specific synchronous formalisms and the associated software tools for program simulation, compiling, and verification. They will support our general claim that synchronous programming opens a new path towards powerful, rigorous, and usable methodologies for reactive and real-time programming.

ACKNOWLEDGEMENTS : the authors are indebted to several reviewers who pointed out misleadings and obscure claims in the first version ; they would like to thank especially Edmund Clarke for his careful reading and criticism of the manuscript.

References

[1] CSML, see this special issue.

[2] ESTEREL, see this special issue.

[3] STATECHARTS, see this special issue.

[4] LUSTRE, see this special issue.

[5] SIGNAL, see this special issue.

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

[7] R. ANDRÉ-OBRECHT, "A new statistical approach for the automatic segmentation of continuous speech signals", *IEEE Trans. on ASSP*, 36-1, 29-40, 1988.

[8] A. BENVENISTE, P. LE GUERNIC, "A denotational theory of synchronous communicating systems", to appear in *Information and Computation.*

[9] G. BERRY, "Real Time Programming: Special Purpose Languages or General Purpose Languages", 11th IFIP Wold Congress 1989, San Francisco.

[10] G. BERRY, "A Hardware Implementation of Pure Esterel", Proc. Workshop on Formal Methods in VLSI Design, Miami, 1991.

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

[12] P. CASPI, "Clocks in Data–flow languages", to appear in *Theoretical Comp. Sc.*, 1990.

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

[14] D. HAREL AND A. PNUELI, "On the Development of Reactive Systems" in *Logics and Models of Concurrent Systems*, NATO ASI Series, Vol. 13 (K. R. Apt, ed.), Springer-Verlag, New York, 1985, pp. 477-498.

[15] G. BERRY AND S. MOISAN AND J-P. RIGAULT, "Esterel: Towards a Synchronous and Semantically Sound High Level Language For Real Time Applications", Proc. IEEE Real Time Systems Symposium, 1983.

[16] J-L. BERGERAND AND P. CASPI AND N. HALBWACHS AND D. PILAUD AND E. PILAUD, "Outline of a Real-Time Data-Flow Language", 1985 Real-Time Symposium, San Diego, 1985.

[17] P. LE GUERNIC, A. BENVENISTE, P. BOURNAI, T. GAUTIER, "SIGNAL : a data-flow oriented language for signal processing", *IEEE Transactions on ASSP*, ASSP-34 No 2, 362-374, 1986.

[18] D. HAREL, H. LACHOVER, A. NAAMAD, A. PNUELI, M. POLITI, R. SHERMAN, A. SHTUL-TRAURING AND M. TRAKHTENBROT, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering* **16** (1990), 403–414.

[19] E.A. LEE, D.G. MESSERSCHMITT, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Trans. on Computers*, Jan. 1987, C-36(2).

[20] E.A. LEE, "Consistency in Data–Flow graphs", Research Report UCB/ERL M89/125, Electronics Research Lab., College of Eng., U.C. Berkeley, 1989, to appear in *IEEE Trans. on Parallel and Distributed Systems.*

[21] A.V. OPPENHEIM, R.W. SCHAFER, *Discrete-time signal processing,* Prentice Hall, 1989.

[22] J. PICONE, "Continuous Speech Recognition Using Hidden Markov Models," *IEEE-ASSP Magazine,* vol 7 Nr 3, 26-41, 1990.

[23] A. PNUELI, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends", in *Current Trends in Concurrency* (de Bakker et al., eds.), Lecture Notes in Comput. Sci., Vol.224, Springer-Verlag, Berlin, 1986, pp.510–584.

[24] W. REISIG, *Petri Nets,* Springer, N-Y, 1985.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

1991

PI 570 **DESIGN DECISION FOR THE FTM : A GENERAL PURPOSE FAUTL TOLERANT MACHINE**
Michel BANATRE, Gilles MULLER, Bruno ROCHAT, Patrick SANCHEZ
Janvier 1991, 30 pages

PI 571 **ANIMATION CONTROLEE PAR LA DYNAMIQUE**
Georges DUMONT, Parie-Paule GASCUEL, Anne VERROUST
Février 1991, 84 pages

PI 572 **MULTIGRID MOTION ESTIMATION ON PYRAMIDAL REPRESENTATIONS FOR IMAGE SEQUENCE CODING**
Nadia BAAZIZ, Claude LABIT
Février 1991, 48 pages

PI 573 **A SURVEY OF TREE-TRANDUCTIONS**
Jean-Claude RAOULT
Février 1991, 18 pages

PI 574 **THE OPTIMAL ADAPTIVE CONTROL USING RECURSIVE IDENTIFICATION**
Anatolij B. JUDITSKY
Février 1991 - 26 pages

PI 575 **MANUEL SIGNAL**
Patricia BOURNAI, Bruno CHERON, Bernard HOUSSAIS, Paul LE
Paul LE GUERNIC
Février 1991, 84 pages

PI 576 **AN INFORMATION BASED RELIABILITY PREDICTOR FOR SYSTEMS IN OPERATIONAL PHASE**
Kamel SISMAIL
Février 1991 - 22 pages

PI 577 **MULTISCALE STATISTICAL SIGNAL PROCESSING AND RANDOM FIELDS ON HOMOGENEOUS TREES**
Albert BENVENISTE, Michèle BASSEVILLE, Ramine NIKOUKHAH, Alan S. WILLSKY, Ken C. CHOU
Mars 1991 - 18 pages

PI 578 **TOWARDS A DECLARATIVE METHOD FOR 3D SCENE SKETCH MODELING**
Stéphane DONIKIAN, Gérard HEGRON
Mars 1991 - 22 pages

PI 579 **SYSTEMES MARKOVIENS DISCRETS STATIONNAIRES ET APPLICATIONS**
Jean PELLAUMAIL
Mars 1991 - 284 pages

PI 580 **DESCRIPTION ET SIMULATION D'UN SYSTEME DE CONTROLE DE PASSAGE A NIVEAU EN SIGNAL**
Bruno DUTERTRE, Paul LE GUERNIC
Mars 1991 - 66 pages

PI 581 **THE SYNCHRONOUS APPROACH TO REACTIVE AND REAL-TIME SYSTEMS**
Albert BENVENISTE
Avril 1991 - 36 pages

PI 582 **PROGRAMMING REAL TIME APPLICATIONS WITH SIGNAL**
Paul LE GUERNIC, Thierry GAUTIER, Michel LE BORGNE, Claude LE MAIRE
Avril 1991 - 36 pages

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

ISSN 0249-6399