Synchronous Modelling of Asynchronous Systems

Nicolas Halbwachs, Siwar Baghdadi

Vérimag, Grenoble - France

Abstract. We investigate the use of synchronous languages to model partially asynchronous systems. Through the use of intermittent activations and assertions, synchronous processes can be composed according to various asynchronous models. Common synchronization and communication mechanisms can be described. The resulting descriptions can be simulated and verified, using classical tools of synchronous programming.

1 Introduction

Synchronous programming [IEE91] is recognized as a very convenient paradigm for designing reactive systems. This success is due to several advantages: synchronous languages [LGLL91,HCRP91,BG92] enjoy clean formal semantics, synchronous parallel composition is an invaluable tool for structuring programs and reducing programming complexity, synchronous programs can be efficiently and safely implemented, usual verification methods [HLR92,LDBL93,JPV95,Bou98] work in general better on synchronous programs, etc.

Now, it is clear also that synchronous programming is not the panacea. There is a need for combining synchrony and asynchrony, from two opposite points of view: on one hand, to introduce synchrony in areas dominated by distributed systems, and on the other hand, in synchronous systems, to take into account the intrinsic asynchrony of interfaces and distributed implementations.

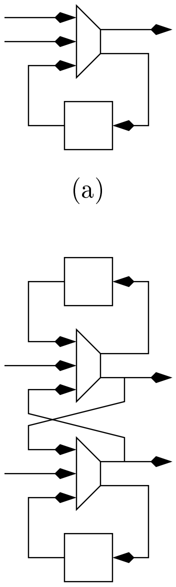
In the domain of distributed systems, and in particular, in telecommunications, people are naturally used with asynchronous languages (e.g., SDL [ITU94]). Now, they use the same languages for programming centralized parts (e.g., feature management), which would be obviously easier with synchronous languages. This shows an obvious need for programming this kind of systems as "*Globally Asynchronous, Locally Synchronous Systems*" (GALS), i.e., synchronous processes composed with each other in an asynchronous way.

On the other hand, asynchrony cannot be completely ignored in synchronous programming. First, the boundaries of a synchronous systems have to deal with asynchronous features, like sampling asynchronous signals [Cas01]. Also, reactive systems are often implemented on distributed architectures. When this implementation is done by hand, one needs some way of modelling the whole system, for early simulation and validation [CS00]. Even in case of automatic code distribution [CGP94,BCT99], there is a need for validating the resulting implementation, which cannot be strictly equivalent to the centralised one.

Several formalisms have been proposed which combine synchronous and asynchronous primitives (e.g., [AH99]). More specifically, the concept of GALS has been investigated in several models and tools [BCG99,BS98,BCG$^+$97]. In this paper, we investigate the use of synchronous languages as they are, to model asynchronous systems, allowing, thus, the modelling of GALS. This is nothing but a systematic presentation of what many people do (e.g., [BS01]) for simulating asynchronous systems with synchronous languages: synchronous processes are activated by more or less related conditions (or clocks), and non determinism is handled through additional inputs. After a formal definition of synchronous machines (Section 2), we formalise sporadic activation of synchronous processes (Section 3), and we investigate various synchronization and communication mechanisms (Section 4). Obvious advantages are that we get formal semantics of GALS for free, and that this synchronous modelling does not increase the size of the models.

2 Synchronous machines

In this section, we give a general formalization of synchronous machines, independently of any synchronous language. The notion we have in mind is a straightforward generalization of synchronous circuits (Mealy machines) to work with arbitrary data-types: such a machine has a memory (a state), and a combinational part, computing the output and the next state as a function of the current input and state (Fig. 1.a). These machines can be composed in parallel, with possible "plugging" of one's outputs into the other's inputs (Fig. 1.b), as long as these wirings don't introduce any combinational loop.



(a)

(b)

Figure 1.
Synchronous machines and their composition

Let *Var* be a set of variables, each variable $v \in Var$ taking its values in a domain D_v. If $X \subseteq Var$, a *valuation* of X is a function associating with each $v \in X$ its value $\in D_v$. We note $\mathcal{V}(X)$ the set of valuations of X.

Definition 1 (Synchronous machines) *Let $\mathcal{I}$, $\mathcal{O}$, $\mathcal{S}$, be three disjoint subsets of Var. A synchronous machine with input variables $\mathcal{I}$, output variables $\mathcal{O}$, and state variables $\mathcal{S}$, is given by an initial state $s_0 \in \mathcal{V}(\mathcal{S})$, an output function $\omega : \mathcal{V}(\mathcal{S}) \times \mathcal{V}(\mathcal{I}) \mapsto \mathcal{V}(\mathcal{O})$ and a next-state function $\sigma : \mathcal{V}(\mathcal{S}) \times \mathcal{V}(\mathcal{I}) \mapsto \mathcal{V}(\mathcal{S})$.*

The machine starts from the initial state s_0. In a given state s, it deterministically reacts to an input valuation i by returning the output valuation

$o = \omega(s,i)$ and moving to the state $\sigma(s,i)$. We will often consider the functions ω and σ as vectorial functions, $\omega_v(s,i)$ (resp. $\sigma_v(s,i)$) being the value $\omega(s,i)(v)$ (resp., $\sigma(s,i)(v)$) associated with v by the valuation $\omega(s,i)$ (resp., $\sigma(s,i)$).

Definition 2 (Dependence) *We will say that the output $v(\in \mathcal{O})$ of a machine depends on the input $w(\in \mathcal{I})$, if the function ω_v depends*[1] *on w.*

Definition 3 (Synchronous composition)
Let $M_p = (\mathcal{I}_p, \mathcal{O}_p, \mathcal{S}_p, s0_p, \omega_p, \sigma_p), (p = 1, 2)$, be two synchronous machines, satisfying:

- $\mathcal{S}_p \cap (\mathcal{I}_q \cup \mathcal{O}_q \cup \mathcal{S}_q) = \emptyset, p = 1, 2, q = 2 - p$ *(state variables are local)*
- $\mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset$ *(disjoint outputs)*
- *for each $v_1 \in \mathcal{O}_1 \cap \mathcal{I}_2$ and each $v_2 \in \mathcal{O}_2 \cap \mathcal{I}_1$, either v_1 does not depend on v_2 (in M_1), or v_2 does not depend on v_1 (in M_2) (no combinational loop)*

Then, their synchronous composition $M_1 \times M_2$ is a synchronous machine, with

- $\mathcal{I}_\times = \mathcal{I}_1 \setminus \mathcal{O}_2 \cup \mathcal{I}_2 \setminus \mathcal{O}_1$ *as input variables*
- $\mathcal{O}_\times = \mathcal{O}_1 \cup \mathcal{O}_2$ *as output variables*
- $\mathcal{S}_\times = \mathcal{S}_1 \cup \mathcal{S}_2$ *as state variables*

and the output $\omega_\times$ and next-state $\sigma_\times$ functions are defined as follows: let i be a valuation of $\mathcal{I}_\times$, (s_1, s_2) be a valuation of $\mathcal{S}_\times$. Let us define conjointly[2] *i_1, i_2, valuations of $\mathcal{I}_1, \mathcal{I}_2$, respectively, as follows*

$$\text{for } p = 1, 2, q = 2 - p,\ i_p(v) = \begin{cases} \omega_q(s_q, i_q)(v) & \text{if } v \in \mathcal{O}_q \\ i(v) & \text{otherwise} \end{cases}$$

Then,

$$\omega_\times((s_1, s_2), i)(v) = \begin{cases} \omega_1(s_1, i_1)(v) & \text{if } v \in \mathcal{O}_1 \\ \omega_2(s_2, i_2)(v) & \text{if } v \in \mathcal{O}_2 \end{cases}$$

$$\sigma_\times((s_1, s_2), i)(v) = \begin{cases} \sigma_1(s_1, i_1)(v) & \text{if } v \in \mathcal{S}_1 \\ \sigma_2(s_2, i_2)(v) & \text{if } v \in \mathcal{S}_2 \end{cases}$$

3 Sporadic activation and nondeterminism

It is well-known, since [Mil81], that asynchrony can be simulated in a synchronous model, by letting processes non-deterministically "stutter", or "stay silent". As a matter of fact, most synchronous languages provide a way of "blocking" or "freezing" a process:

[1] Here, we consider a syntactic notion of dependence. Of course, the whole problem of *causality* [Mal93,HM95,SBT96] is hidden here, but it will not be addressed in this paper.

[2] Notice that this definition makes sense because of the absence of combinational loop.

– in Esterel, the "suspend" statement has been identified [Ber93] as a very basic primitive: "suspend P when S" prevents P from reacting when the signal S is present.
– in Lustre and Signal, the concept of clock is available to define when a process is triggered: according to the data-flow nature of these languages, an operator is only activated at the arrival rate of its operands. By "filtering" the input flows of a process according to a clock, one can rule out the activation rate of the process.
– in synchronous circuits also, this would be done by disabling the clock.

Let us formalize this notion in our model of synchronous machines: we define an operation consisting of adding a Boolean input to a machine, which freezes the state when it is 0. Now, we have to define the output when the machine is not activated. There are several solutions:

– In Esterel and Signal, the machine is "silent", meaning that the outputs take a special "absent" value.
– In Lustre, the outputs are frozen to their last taken values (through the use of the "current" operator). This may be expensive in memory (or in number of states, for verification).
– In circuits, the combinational part would keep computing the outputs, as a function of the frozen state and the current inputs. This is not a very intuitive behavior.

In fact, this point is not very important, since we will manage not to make use of the outputs of a process when it is not activated. Let us choose the Esterel and Signal solution, which is both clean and inexpensive:

Definition 4 (Activation) *Let $M = (\mathcal{I}, \mathcal{O}, \mathcal{S}, s0, \omega, \sigma)$ be a synchronous machine, and let $c \in Var \backslash (\mathcal{I} \cup \mathcal{O} \cup \mathcal{S})$ be a variable not involved in M, and such that $D_c = \{0, 1\}$. Then the activation of M according to c is a synchronous machine $c \triangleright M = (\mathcal{I} \cup \{c\}, \mathcal{O}, \mathcal{S}, s0, \omega^{\triangleright}, \sigma^{\triangleright})$, where*

$$\sigma^{\triangleright}(s, (i, c)) = \begin{cases} \sigma(s, i) & \textit{if } c = 1 \\ s & \textit{otherwise} \end{cases} \qquad \omega^{\triangleright}(s, (i, c)) = \begin{cases} \omega(s, i) & \textit{if } c = 1 \\ \perp & \textit{otherwise} \end{cases}$$

c will be called the activation condition *of $c \triangleright M$.*

The second issue for modelling asynchrony, is non-determinism. Not only the non-deterministic scheduling of asynchronous processes must be modelled, but we will often have to model non-deterministic processes.

An obvious, and very convenient way of introducing non-determinism, is by using additional arbitrary inputs, with arbitrary values. For instance, non-deterministic scheduling will be modelled as follows: if each machine M_p has its own activation condition c_p, which is an input to the whole system, then the composition $(c_1 \triangleright M_1) \times (c_2 \triangleright M_2) \times \ldots$ will behave as the asynchronous product of the machines.

Now, we have to be careful about the communications in this asynchronous composition (since we decided that the outputs of a non-activated machine is the undefined value $\perp$). Moreover, in general, we want some synchronization between the machines. So, the issues of communication and synchronization must be addressed.

4 Communication and Synchronization

Many mechanisms have been proposed for the communication and synchronization in concurrent systems, and [Ber89] noticed that none of them can replace all the others. Here, we will show, on some examples, how these mechanisms can be modelled in our framework. Basically, we will use two ways for describing communication and synchronization:

- Communication and synchronization media will be modelled as synchronous processes. As far as activation is concerned, these processes need to be activated at least as often as the main processes they interact with. In general, we will consider that they are always active.
- Since non-deterministic scheduling is represented by means of auxiliary inputs, it can be restricted by means of constraints on these inputs. Once again, we can benefit of a feature which exists in most synchronous languages, which allows arbitrary *assumptions* to be expressed about the inputs of a program: "*relations*" in Esterel, "*assertions*" in Lustre, "*synchro*" in Signal.

4.1 Shared memory

Let us start with a very simple model of time-sharing systems with shared memory. There is an assumption of *atomicity*: processes perform atomic actions in turn. In our setting, it means that the activation conditions are exclusive. Communication is by shared memory: a memory cell is a process, taking as inputs a "write" signal, and a "written" value, and continuously returning as output the last written "value", which is also its only state variable:

$$\begin{aligned}&\mathsf{value} = \mathsf{state}\\&\mathsf{state}' = \mathsf{if\ write\ then\ written\ else\ state}\end{aligned}$$

Example 1: Peterson mutual exclusion algorithm. As an example, Fig. 2 shows an Esterel program simulating the Peterson mutual exclusion algorithm. The program is written in pure Esterel (no values, only pure signals). Each process is an instance of the module "process", which implements the algorithm. Each atomic action is performed within a single step. An additional input "go" is used to control the non-deterministic arrival in and departure from the critical section. The three shared memory cells are instances of the module "cell", which implements a Boolean cell with pure signals. Finally, the main module "peterson" instanciates the two processes, activated by input signals "c0" and "c1" (assumed to be exclusive, by means of a "relation"), and the three memory cells (which are always active). If we generate the automaton of this program, we obtain exactly the classical one, meaning that our modelling does not introduce any extra complexity for verification. The mutual exclusion property is straightforwardly verified, using, e.g., Xeve [Bou98].

Notice that the exclusivity of the processes reactions could be weakenned, by assuming only that the "*write*" signals on the same memory (i.e., `setx` and `resetx`) are exclusive. This is not permitted by Esterel relations (which are restricted to input signals), but it would be possible with Lustre assertions.

4.2 Rendez-vous

Another popular mechanism is the synchronization and communication by rendez-vous. Here, both processes involved in a rendez-vous should share a step, so the activation conditions are not exclusive. We implement the rendez-vous as follows: both processes emit a Boolean variable (or maintain a pure signal) telling whether they are ready. A handler simultaneously returns them an acknowledgement when they are both ready and active. It's the responsibility of the processes to go through the rendez-vous when they receive the acknowledgement. Notice that this mechanism involves an "instantaneous dialogue" between the processes and the handler, and thus makes use of the full power of synchrony.

Example 2: Milner's cycler in Esterel
As a first example, we consider a very simplified version of the "cycler" proposed by R. Milner in [Mil80]: n processes, organized in a ring, have to be scheduled so that each of them makes a step in turn. For that, each process P_i first synchronizes with its predecessor in the ring, to get the grant, then perfoms its own step (emitting a signal a), and then synchronizes with its successor to transmit the grant. A special "starter" process first gives the grant to P_0.

Fig. 4 shows our solution, in Esterel, for two processes. Each process starts by a rendez-vous (with the other process) using a signal "`read_g`", and waiting for the acknowledgement "`ack_g`", then it emits its own signal "`a`" and starts a rendez-vous using the signals "`send_G`" and "`ack_G`". The starter is just a degenerate version of a process. A rendez-vous handler emits an aknowledgement whenever its "`read`" and "`send`" input signals occur simultaneously. Finally, the main program instanciates the starter and two processes — non deterministically activated by the input signals "`c0`", "`c1`", and "`c2`", respectively, on which there is no specified relation — and the two handlers.

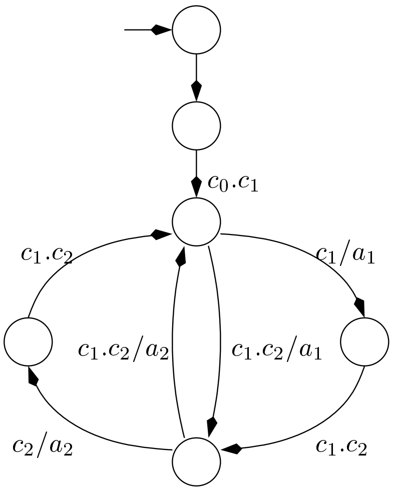


Figure 3: The automaton of the cycler

```
module process:
input d, dprime, x, go;
output sd, rd, cx, crit;
loop
   await go;
   emit sd; await tick;
   emit cx; await tick;
   await [x or not dprime];
   do sustain crit upto go;
   emit rd; await tick;
end.
```

```
module cell:
input set, reset;
output cell;
loop
   await set;
   do sustain cell upto reset;
end.
```

```
module peterson:
input c0, c1, go0, go1;
output crit0, crit1;
relation c0 # c1;
signal d0, d1, x0, x1,
       set0, reset0, set1, reset1, setx, resetx in
[
   suspend
      run process [ signal d0/d, d1/dprime, x1/x, go0/go,
                           set0/sd, reset0/rd, resetx/cx, crit0/crit ]
   when [not c0];
||
   suspend
      run process [ signal d1/d, d0/dprime, x0/x, go1/go,
                           set1/sd, reset1/rd, setx/cx, crit1/crit ]
   when [not c1];
||
   run cell [ signal set0/set, reset0/reset, d0/cell ]
||
   run cell [ signal set1/set, reset1/reset, d1/cell ]
||
   run cell [ signal setx/set, resetx/reset, x1/cell ]
||
   every [not x1] do emit x0 end
]
end.
```

Fig. 2. Peterson's algorithm in Esterel

Again, we get exactly the expected automaton (except for the spurious initial state, systematically introduced by the Esterel compiler), shown by Fig. 3.

Example 3: the alternating bit in Lustre Here we consider rendez-vous with communication. In Lustre, the writter process will make use of two variables: a Boolean variable signaling a request for communication, and another variable carrying the value to be passed. The reader process signals its readiness with a Boolean variable. The rendez-vous handler responds whenever both partners are active, and their Boolean variables (request, readiness) are true:

```
node Handler(act1, act2, ready1, ready2: bool)
  returns (rendez_vous: bool);
let
  rendez_vous = act1 and ready1 and act2 and ready2;
tel
```

In the considered version of the alternating bit, both the sender and the receiver process involved in the protocol have two states: in state N the sender acquires a new message (resp., the receiver transmits a new message), while in state S the sender tries to send the current message (resp., the receiver tries to receive a new message). We encode these automata by means of Boolean equations.

The sender elaborates the sequence of messages to be sent (here, for simplicity, the list of natural numbers!), it receives the grant for rendez-vous from the line to (SR) and from (RS) the receiver, together with the acknowledgement bit from the receiver (which it may only read when the rendez-vous RV_RS occurs). It outputs the current message and its associated bit, and the requests for rendez-vous with the line to and from the receiver. Its stays in state N only one step, in which it changes its bit and takes a new message (i.e., increment its value), then enters the state S, where it maintains requests for rendez-vous for sending the message and reading the acknowledgement, until it receives an acknowlegement bit equal to the bit sent.

```
node Sender (RV_SR, RV_RS: bool; bit_ack: bool)
returns (m_sent: int; bit_sent: bool; W_SR, R_RS: bool);
var N, S: bool (* the state *); mybit: bool;
let
   (m_sent,bit_sent) = (0,false) ->
                       if N then (pre(m_sent)+1,not pre(bit_sent))
                            else (pre(m_sent),pre(bit_sent));
   W_SR = S; R_RS = S;
   N = true -> pre(S and RV_RS and bit_ack=bit_sent);
   S = false -> pre(N or not RV_RS or bit_ack<>bit_sent);
tel
```

The receiver node is symmetrical. Each part of the line (from the sender to the receiver, and conversely) is modelled by a simple node, which has an

```
module proc:
input ack_g, ack_G;
output a, read_g, send_G;
loop
  weak abort sustain read_g when immediate ack_g;
  await tick; emit a;
  weak abort sustain send_G when immediate ack_G;
  await tick
end.

module handler:
input read, send;
output ack;
every [read and send] do emit ack end.

module starter:
input ack_G;
output send_G;
weak abort sustain send_G when ack_G .

module cycler:
input c0, c1, c2;
output a1, a2;
signal ack1, ack2, r1, r2, g1, g2 in
  suspend
    run proc [signal ack1/ack_g, ack2/ack_G, a1/a, r1/read_g, g2/send_G]
  when [not c1]
||
  suspend
    run proc [signal ack2/ack_g, ack1/ack_G, a2/a, r2/read_g, g1/send_G]
  when [not c2]
||
  suspend
    run starter [signal  ack1/ack_G, g1/send_G]
  when [not c0]
||
  run handler [signal r1/read,  g1/send, ack1/ack]
||
  run handler [signal r2/read, g2/send, ack2/ack]
end.
```

Fig. 4. Milner's cycler in Esterel

additional input used to non-deterministically lose messages. Here is the line from the sender to the receiver:

```
node Line (RV_in, RV_out: bool; m_in: int; bit_in: bool; lose: bool)
returns (R_in, W_out: bool; m_out: int; bit_out: bool)
let
  R_in = true; W_out = true;
  (m_out,bit_out) = (m_in,bit_in) ->
                        if RV_in and not lose then (m_in,bit_in)
                        else (pre(m_out),pre(bit_out));
tel
```

It permanently asks for input (R_in) and output (W_out) rendez-vous, and changes the transmitted values of the message (m_out) and the bit (bit_out) whenever the input rendez_vous (RV_in) occurs and the transmission is not lost.

If we connect the four processes (Sender, Receiver, Line_SR, Line_RS), activated with arbitrary activation condition, and connected through 4 rendez-vous handlers, we get the whole program, which behaves as expected, and can be verified with no extra cost.

4.3 Other communication/synchronization mechanisms

Other classical communication mechanisms — like buffers, FIFO [dSB98],... — can be programmed in a similar way. Another game is to play with the control we have on the activation conditions to express our knowledge about the processes speed: for instance, in [CS00], the authors introduce the concept of *quasi-synchrony*. In this model, processes are periodically activated, with almost the same periods. The only constraint is that, for any two processes in the system, there are at most two activations of each of them between two consecutive activations of the other. Since this is the only synchronization constraint between the processes, which otherwise communicate by shared memory, it must be taken into account in any simulation or validation of the system. This constraint can be easily modelled by assertions on the activation conditions (and this is exactly what is done in [CS00]).

5 Conclusion

In this paper we tried to systematize and popularize a way of modelling asynchronous systems by means of synchronous languages, for simulation and validation purposes. This approach is not really new: for instance, it is eactly the way [BS01] simulates Multi-clock Esterel in basic Esterel. However, we think that it deserves better attention in the community of asynchronous system modelling and validation.

As a matter of fact, the approach has several advantages: it provides standard semantics to various asynchronous composition mechanisms, it allows synchronous and asynchronous parts of a system to be simply, cleanly, and systematically merged. Moreover, it allows a significant reduction of the validation cost:

synchronous description of synchronous parts often involves a drastic reduction of the number of states, and experience shows that no extra cost is introduced by (careful) synchronous modelling of asynchronous composition.

This experience inspires also some conclusions about synchronous languages. While the sporadic activation exists in all languages, some care should be paid to the way it is defined: in Lustre, the fact that the outputs of a suspended process are frozen can result in many extra states, which can be expensive in verification. The use of a default value should be preferred. On the other hand, assertions appear to be a very important feature, for restricting asynchrony and non-determinism, and their full power is needed: we noticed that Esterel relations, in their current version, are too weak for some purposes.

The immediate perspectives of this work are twofold:

- A library of standard synchronization and communication modules should be defined, to help the user in describing GALs.
- The automatic translation of such description towards asynchronous or distributed implementation should be considered.

References

[AH99] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[BCG99] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99*. LNCS 1664, Springer Verlag, 1999.

[BCT99] A. Benveniste, P. Caspi, and Stavros Tripakis. Distributing synchronous programs on a loosely synchronous, distributed architecture. Research Report1289, Irisa, December 1999.

[Ber89] G. Berry. Real time programming: Special purpose or general purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.

[Ber93] G. Berry. Preemption and concurrency. In *Proc. FSTTCS 93*, Lecture Notes in Computer Science 761, pages 72–93. Springer-Verlag, 1993.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[Bou98] A. Bouali. Xeve: an Esterel verification environment. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[BS98] G. Berry and E. Sentovich. Embedding synchronous circuits in GALS-based systems. In *Sophia-Antipolis conference on Micro-Electronics (SAME 98)*, October 1998.

[BS01] G. Berry and E. Sentovich. Multiclock esterel. In *Correct Hardware Design and Verification Methods, CHARME'01*, Livingston (Scotland), September 2001. LNCS 2144, Springer Verlag.

[Cas01] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *1st International Workshop on Embedded Software, EMSOFT2001*, Lake Tahoe, October 2001. LNCS 2211.

[CGP94] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, USA, October 1994. ISCA.

[CS00] P. Caspi and R. Salem. Threshold and bounded-delay voting in critical control systems. In *FTRTFT'2000*, Pune, India, September 2000. LNCS 1926.

[dSB98] R. de Simone and A. Bouali. A symbolic representation of asynchronous networks of synchronous processes. In *Tenth International Workshop on Logic Synthesis, IWLS'98*, Lake Tahoe, June 1998.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[HM95] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, Como (Italy), September 1995.

[IEE91] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[ITU94] ITU-T. *Recommendation Z-100. Specification and description language (SDL).* 1994.

[JPV95] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of esterel programs and applications to telecommunication software. In P. Wolper, editor, *7th International Conference on Computer Aided Verification, CAV'95*, Liege (Belgium), July 1995. LNCS 939, Springer Verlag.

[LDBL93] M. Le Borgne, Bruno Dutertre, Albert Benveniste, and Paul Le Guernic. Dynamical systems over Galois fields. In *European Control Conference*, pages 2191–2196, Groningen, 1993.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[Mal93] S. Malik. Analysis of cyclic combinational circuits. In *ICCAD'93*, Santa Clara (Ca), 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer Verlag, 1980.

[Mil81] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edimburgh Univ., 1981.

[SBT96] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *International Design and Testing Conference IDTC'96*, Paris, France, 1996.