Embedded Control: From Asynchrony to Synchrony and Back*

Paul Caspi
caspi@imag.fr

Verimag–CNRS
http://www-verimag.imag.fr/

Abstract. We propose in this paper a historical perspective of programming issues found in the implementation of control systems, based on the author's observations for more than fifteen years, but especially during the Crisys Esprit project. We show that in contrast with the asynchronous tradition of computer scientists, control engineers were naturally led to a synchronous practice that was later formalised and generalised by computer people. But, we also show that, for the sake of robustness and distribution those practitioners had to incorporate some degree of asynchrony in this synchronous approach and we try to comment the resulting programming style.

1 Introduction

The history of computer implementations of control systems is really an interesting one marked by several unexpected accomplishments, among which we can cite:

- The invention, by control engineers, of their own programming languages in place of those designed for them by computer scientists.
- The extraction by computer scientists of those inventions, for long buried into in-house products, and their promotion to the status of a new programming paradigm called "Synchronous programming".
- The use of simulation tools like Matlab/Simulink as programming languages.
- etc.

Having been involved in this history [20], we were aware of this complicated landscape but we did not still know how far it could go. In the

* Paper presented at EMSOFT 2001, First International Workshop on Embedded Software, Lake Tahoe, October 2001 and published as Lecture Notes in Computer Science of Springer Verlag, volume number 2211.

course of an Esprit project (Crisys (97–01)), we had the occasion of observing several achievements in the domain of distributed control systems:

- The Airbus “fly-by-wire” system.
- Schneider’s safety control and monitoring systems for nuclear plants.
- Siemens’ letter sorting machine control,

and several other distributed safety-critical control systems we had previously studied. We had then the surprise of noticing that in most systems, synchronous programming tools are used in an *asynchronous programming style* for quite obvious reasons of robustness, thus returning in some sense to the initial computer science programming styles that had been initially rejected.

This paper intends to trace back the history of this “surprise”. It is organised as follows:

- At section 2 we briefly describe the basic needs of the domain and show how the computer science answers to these needs were at least partially unsatisfactory. Then we show on which grounds practitioners developed their own concepts that were later organised within the “synchronous programming” school of thought.
- Then we show at section 3 how this later programming paradigm is also inadequate when dealing with real-time and distribution.
- Finally, we describe at section 4 the “asynchronous-synchronous” programming style that is used in practice.

2 From Asynchrony to Synchrony

2.1 Basic Needs of the Domain

It has been recognised for long that programming control systems needs to address among others, the following problems:

Account for parallelism. There are two basic reasons for this need: on the one hand (we shall see an example of this question at section 4.1), the control program runs in parallel with the environment it aims at controlling, and studying, synthesising, debugging, testing and formally verifying the control program requires running, in one form or another, some

kind of a model of this environment. Thus the programming language must provide some notion of parallelism. On the other hand, in most cases, the environment has several degrees of freedom that must be controlled in parallel. For instance, in an aircraft, the pitch and the roll must be controlled *at the same time.* This is another reason why the programming language must provide means of naturally describing these activities in parallel.

Provide guaranteed bounds on memory and execution time. Most control systems exhibit hard real-time requirements and are at the same time safety critical ones. It is thus very important that these requirements be met in some sense *by construction* or, at least that their checking be made as easy as possible.

Allow for distribution. Finally most of these systems are distributed ones, for evident reasons of load, location of sensors and actuators and fault-tolerance (redundancy). The programming environment should then also provide facilities for that purpose.

2.2 The Computer Science Answer: Real-Time Kernels and Languages

At the end of the seventies, computer scientist became aware of these requirements and began looking at ways of fulfilling them. Quite uniformly, their proposals were based on the experience they had of programming parallel and distributed activities and most of this experience came from time sharing operating systems. In these systems the main problem was to regulate the concurrent access of several users to computing resources. From this need came the structuring concepts of:

– Synchronisation: semaphores, monitors, sequential processes...
– Communication: shared memory, messages, mail-boxes...
– Synchronisation + communication: queues, rendez-vous

and these structuring concepts were also the ones which structured the proposals in the field of real-time programming, for instance, the programming languages CSP [13], OCCAM, the tasking part of ADA and the many proposals of real-time operating systems.

2.3 The Evolution of Practices

By the same time, practitioners were thinking of moving from their analog controllers to computerised ones. Figure 1 shows a quite general scheme they used for it: it consists of providing a single real-time periodic clock which triggers both analog to digital converters at the input, digital to analog ones at the output and *the computing activity of the computer*. This activity can result from various programming styles, at the source level, but it appears, at the object level, quite uniformly, as a *single program* looking like the one displayed at table 1, and corresponds to what we would call now a periodic synchronous program.

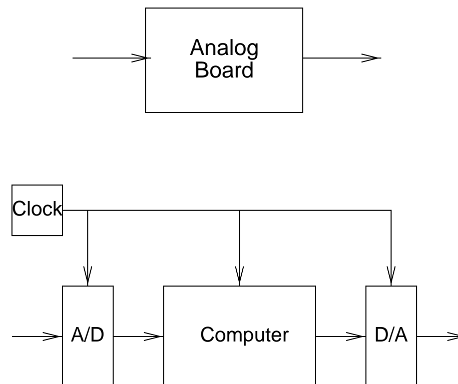


Fig. 1. From analog boards to computers

```
initialize state;

loop each clock tick

   read other inputs;
   compute outputs and state;
   emit outputs

 end loop
```

Table 1. A periodic synchronous program

This kind of implementation has a lot of practical interests:

– First, it perfectly matches the needs for solving, in real-time, the differential equations corresponding to the previous analog boards. It is known that there are many ways of solving differential equations but, if we want to do it in real time, the most practical solution consists of using forward fixed step methods which correspond to the program of table 1.
  It also matches quite closely *the mathematical theory of sampled control systems* [14], a very popular method of control.
– It is also a very simple, safe ant efficient implementation method: There is a single interrupt (the real-time clock ticks) and this interrupt should occur when the computer has finished the loop iteration and is idle. Thus there is no need for context saving. This means that there is very little need for an operating system and that this kind of program can run on a bare machine. This is quite safe if we think that validating an OS is a difficult task and that most standards for safety critical programming like DO178B recommend a limited use of interrupts.
  Furthermore, this programming structure eases the checks for guaranteed bounds on memory and execution time. Bounded memory is obtained by construction if the programming language does not allow for dynamic memory nor recursion. Bounded execution time amounts to checking the worst execution time of an acyclic program.

It is thus an appealing method the more so as safety critical systems are concerned and certification authorities have to be convinced. This is why it has been quite widely applied [6,16,3].

2.4 Generalisation: Synchronous Languages

This kind of practice gave birth to the so-called *synchronous programming school*. The activities of this school consisted essentially of:

– Generalising the concept of clock to any kind of event, (the multiple time scale paradigm) yielding the object code structure of table 2.
– Finding several styles of source code: data-flow [10,2], imperative [4], graphic [12,17,1]. The main shared feature of these different styles is the presence of a parallel construct which allows to address

one of the requirements stated at section 2.1. It must be noticed here that, in order to yield an object code like the one depicted at table 2, this parallel construct has to be *compiled* instead of being *interpreted* like in the concurrent approaches of section 2.2.

- Equipping the approach with efficient compilers, debugging, simulation and formal verification tools [11,5].

```
initialize state;

loop each input event

    read other inputs;
    compute outputs and state;
    emit outputs

end loop
```

Table 2. Synchronous programming

Yet, it should also be noticed that, in practice, most applications of synchronous programming in the control domain are actually periodic ones.

2.5 Milner's SCCS Theory

As for the theoretical foundations of the synchronous approach, we think that it had been *a priori* provided by Robin Milner's SCCS [19]. It was based on the synchronous product of automata and figure 2 shows how this product contrasts with the asynchronous one of CCS. As a the nice feature of this theory, Milner shows that SCCS can simulate CCS and thus that CCS is a *sub-theory* of SCCS. This, in our opinion provides some theoretical support to control practices: this means that synchronous primitives are stronger than and encompass asynchronous ones. If it is so, why should those practitioners have adopted the computer science asynchronous proposals and have limited themselves to weaker primitives?

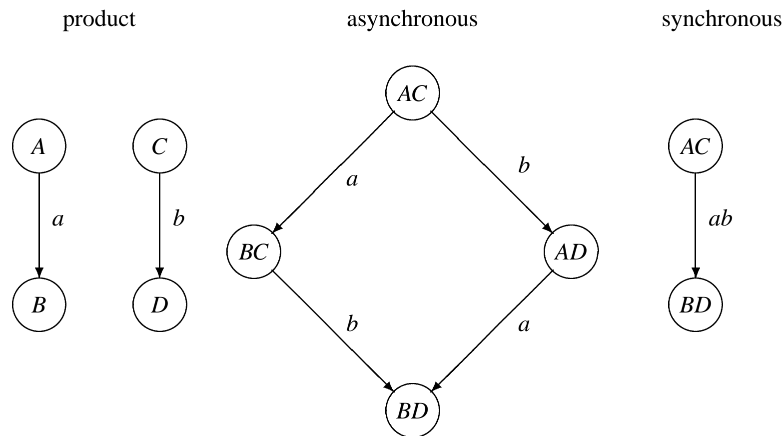


Fig. 2. Asynchronous and synchronous products of automata

2.6 Further Justifications

To this landscape Gérard Berry and the Esterel team added several interesting remarks [4]:

- First they remarked that the synchronous product is deterministic and yields less states than the asynchronous one. This has, in their opinion two valuable consequences: programs are more deterministic and thus yield easier debugging and test and have less state explosion and thus yield easier formal verification.
- They also remarked that a step in the synchronous product corresponds exactly to one step of each of the component automata. This provides a “natural” notion of *logical time* shared by all components which allows an easier and clearer reasoning about time.

We thus see that synchronous programming is endowed with both a rich theory and many pragmatic interests and this seems to justify its being largely used in practice, even by people who don’t know about it. But . . .

3 Some Drawbacks of Synchronous Programming

But the synchronous model is not without problems when applied to control. We discuss here two of them, the first one which is due to hybridity, *i.e.*, the fact that the environment of a control program which provides it

with inputs and receives its outputs does not evolve according to logical time but to real time, and the second one which is due to distribution and amounts to the fact that, in many implementations, control programs do not only sample their environment *but also the other control programs* which it cooperates with. In this sense, the latter is just a generalisation of the former.

3.1 Real-Time is not Logical Time

The phenomenon can be seen at the two boundaries of the control system, *i.e.*, inputs and outputs:

Sampling inputs. Figure 3 shows two possible periodic samplings of the same couple of inputs. In the first one, a and b are seen to raise at the same logical instant while, in the second one, this is not the case. Clearly, a control program should be in some sense insensitive to this sampling phenomenon and, clearly also, synchronous programming does not provide specific means to ensure it.

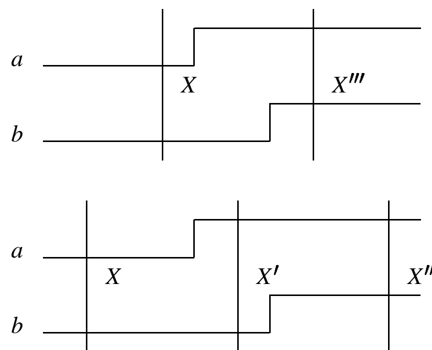


Fig. 3. Sampling is non deterministic

Outputs. Let us consider the following requirement for a control system:

> *y and z should never be true at the same time*

and assume a designer provides you with a solution yielding the chronogram of figure 4(a). Would you be satisfied with this solution, even if he formally proves that it is correct? It is likely that you wouldn't, because, even if the solution is perfectly correct in logical time, it doesn't exclude the possibility of z and y being simultaneously true for some small real-time intervals.

This is why a smart designer is likely to provide you with a solution yielding, for instance, the chronogram of figure 4(b) which ensures you that your desired property holds both in logical and real time.

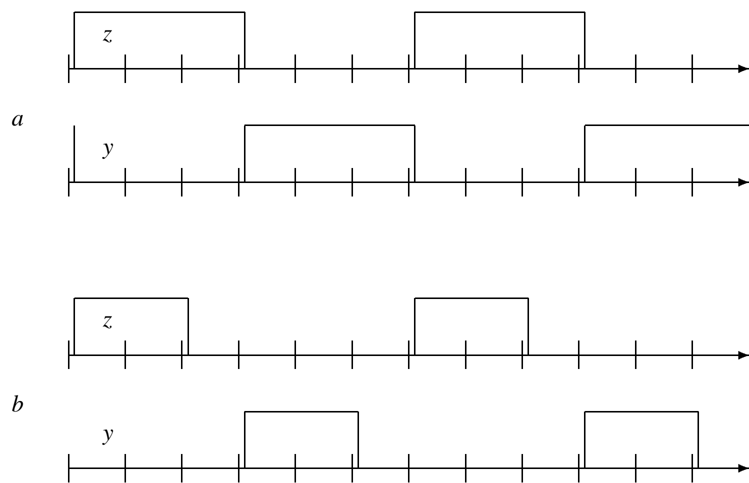


Fig. 4. Non robust (a) and robust (b) mutual exclusion

3.2 Distribution

We have seen at section 2.3 how control engineers moved from analog controllers to computers. But, in general, large control systems like a commercial aircraft flight control or a chemical plant control are made of more than one board and figure 5 shows how, in most cases, they did when dealing with networks of analog boards.

The idea here was to reuse repeatedly the replacement scheme of figure 1, but for the case when two adjacent boards were replaced by

computers. Then the previous analog communication was replaced by serial lines and in further evolutions, by so-called field busses [9].

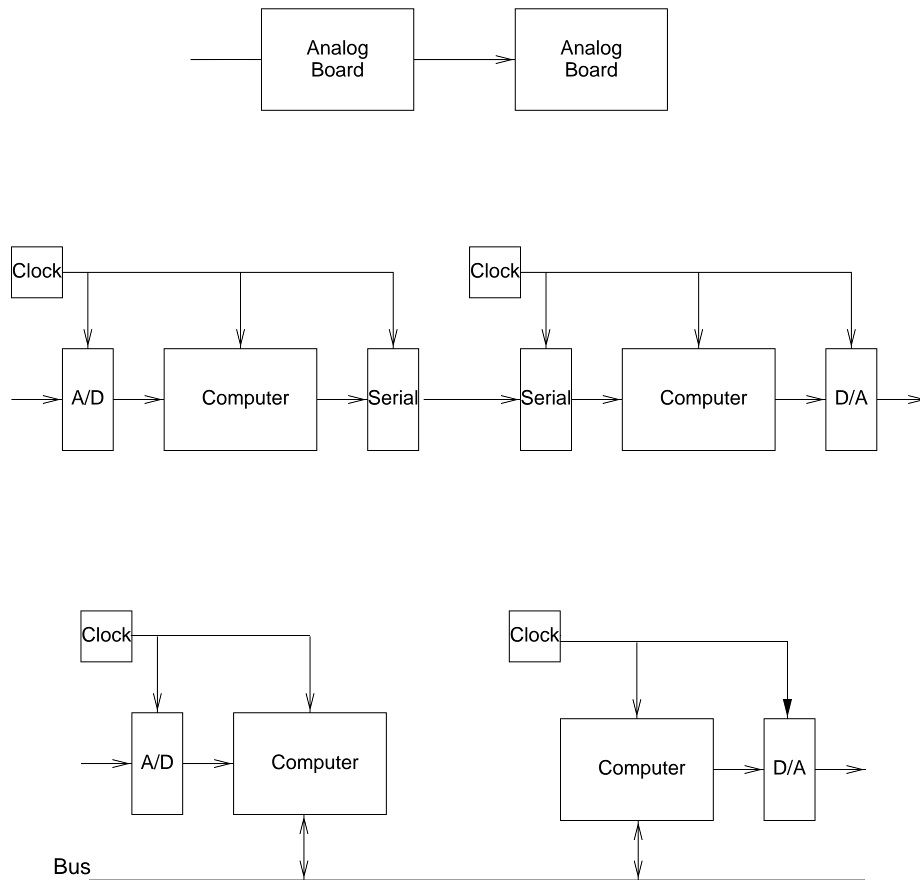


Fig. 5. From networks of analog boards to local area computing networks

This method has many advantages:

- It is modular, *i.e.*, it allows to progressively replace elements, according to the needs, and even to let several technologies cooperate smoothly.
- It can be seen as robust:
  Each computer is a complete and autonomous one, including its own real-time clock and even, possibly, its own power supply.

Communications between computers are based on periodic readings and writings. In some sense, they are similar to the communications between computers and environments, that is, based on periodic sampling and, as it, are non blocking. Thus liveness, at least at low level, is not a problem.
– Finally, it is cost effective, in that it does not need too much specialised hardware. This allows to follow the technological advances at lower cost.

However, it also has its drawbacks. These are summarised at figure 6 which shows a possible behaviour of the real-time clocks of two computers: even if these clocks have been initially set to the same period and phase, this exact situation cannot be maintained in time because no resynchronisation is provided here, in contrast with the so-called *Time Triggered Architecture* [15]. Even if the the periods are assumed to show very limited variations, [1] some problems cannot be avoided:

– Communication errors occur: it may be that two write clocks take place strictly between two read clocks. Then the first write is overwritten and lost. Conversely, data duplication can take place. Moreover, there are non deterministic communication delays. However these delays are bounded: the worst delay occurs when a read clock occurs just before a write clock. Then the written value has to wait another read period before being read.
– Absolute time is lost.
– Finally, this situation amounts to some kind of bounded fairness:

  It is not the case that a component process executes more than twice between two successive executions of another process.

We thus see here that some care has to be taken when using the synchronous paradigm in programming control systems, because some degree of asynchrony has to be incorporated in it for real-time and distribution reasons. In the following section, we shall see how people use to handle it in practice.

[1] In most of the systems considered here, clock periods should be strictly monitored because otherwise even relative time would be lost yielding unrecoverable consequences on system safety [16].

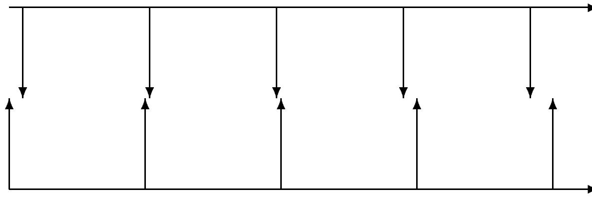


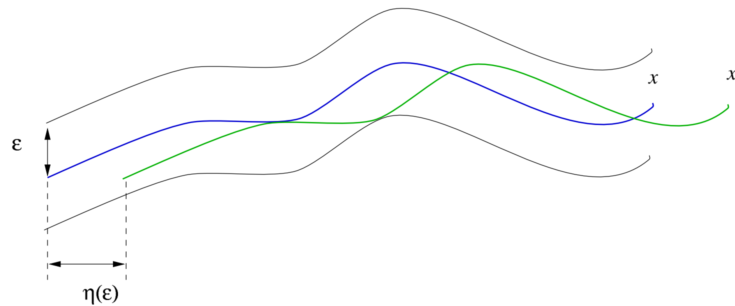
Fig. 6. Two periodic clocks with nearly the same period

4 Asynchronous–Synchronous Programming

Control systems and programs are usually concerned with two types of computations, continuous ones and discrete ones and also with the interaction of both. These will be examined in sequence:

4.1 Continuous Signals and Systems

Figure 7 shows the basic uniform continuity signal property used in this framework. This property says that given an arbitrary maximum error, we can choose a maximum delay such that, staying within this delay ensures staying within this error.



$$\forall \varepsilon > 0, \exists \eta > 0, \forall t, t', |t - t'| \leq \eta_x \Rightarrow |x(t) - x(t')| \leq \varepsilon$$

Fig. 7. An uniformly continuous signal

Now, this property can be used in conjunction with the corresponding property of systems, depicted at figure 8, which says that a system is uniformly continuous if, given an arbitrary maximum output error, there

exists a maximum input error such that, if the input stays within the latter, the output will stay within the former.

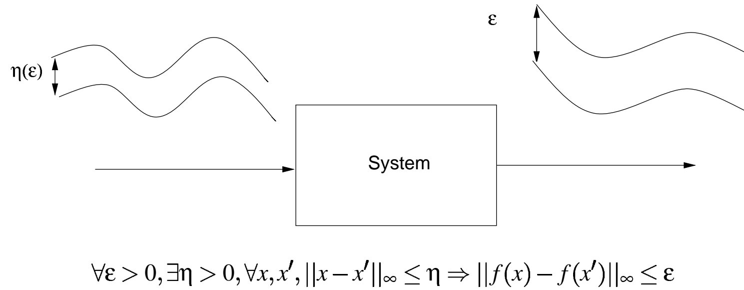


Fig. 8. An uniformly continuous system

Both properties then combine nicely thanks to the following theorem:

Theorem 1. *An uniformly continuous and time-invariant system, fed with uniformly continuous signals, outputs uniformly continuous signals.*

Thus the property also propagates through acyclic networks of such systems connected by bounded delays in such a way that one can find bounds on delays and on input errors such that output errors remain within given bounds.

This seems to give us a satisfactory theory of robust computations over continuous signals. However, let us note here that the landscape is a bit more complex because even very simple controllers like for instance very popular PIDs do not enjoy the uniform continuity property.[2] This situation is recovered within the framework of closed-loop system: unstable controllers are used in order to stabilise the environment in such a way that the closed-loop behaviour of the system appears stable and hence uniformly continuous (Figure 9).

4.2 Uniform Bounded Variability Signals and Combinational Systems

When we move from continuous to discrete signals, we face the problem that errors cannot be given arbitrary small values. The idea is then to only reason about delays.

[2] This is due to the integral part which accumulate errors without ever forgetting them.

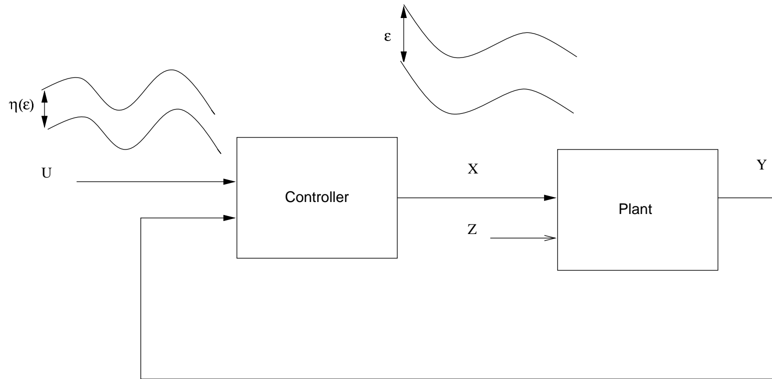


Fig. 9. The closed-loop system computes uniformly continuous signals

Figure 10 illustrates the concept of uniform bounded variability which appears the analogue of uniform continuity for discrete signals. A signal has this property if there exists a least *stable time* between two successive discontinuity points.

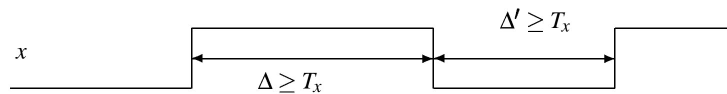


Fig. 10. Uniform bounded variability

In this context, one could expect that combinational functions play the part of uniformly continuous systems in the continuous framework. Unfortunately, this is not the case because delays do not combine nicely as errors do. In particular, as soon as we deal with functions of several variables, we face the problem that independent delays on tuples do not yield delayed tuples. This situation is illustrated at figure 11 where we can see that the value $x' = 0, y' = 1$ does not correspond to any value in the original tuple.

A solution to this problem can be found in the confirmation functions [8] shown at table 3 which are used in order to transform incoherent delays into coherent ones. The idea of this function is that, if we know

bounds on the delays for each component of a tuple and if these components do not vary too fast (thanks to uniform bounded variability), then, each time a component of the tuple changes, we wait at least for some delay before outputting the change: if the other component of the tuple has not changed meanwhile, we can output the tuple value and be sure that this tuple is a delayed value of the original tuple. This can be summarised as:

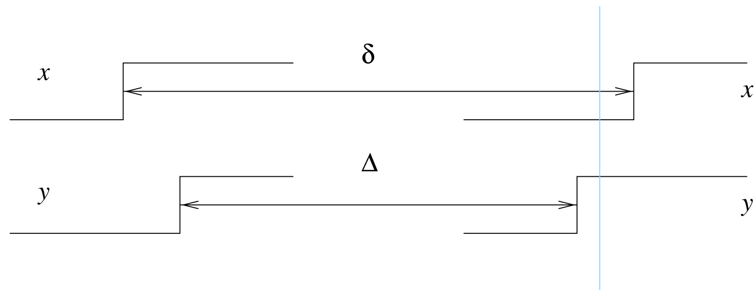


Fig. 11. Delays on tuples do not yield delayed tuples

Theorem 2 (Confirmation). *Given x', y' two bounded delay images of two uniform bounded variability signals x and y, one can find bounds on the delays, on the clock period and on nmax such that $Confirm_{nmax}(x', y')$ is a bounded delay image of the tuple (x, y).*

Here also, we can prove that the uniform bounded variability property propagates through acyclic networks of combinational functions connected by bounded delays and conveniently guarded by confirmation functions in such a way that one can find bounds on the delays, confirmation function parameters, and stable times of the inputs such that the outputs have given stable times and delays (with respect to the ideal undelayed computations). In some sense, theorem 2 appears as an analogue of theorem 1

4.3 Robust Sequential Systems

Unfortunately, this approach does not work as soon as we consider sequential systems. Figure 12 illustrates the critical race phenomenon [7]

```
xp := x0; xpp := x0; n := 0;

loop each clock tick

    read x ;
    if x = xp
    then  if n >= nmax
          then xpp := x; n := 0
          else n := n+1
          end if
    else  xp := x; n := 0
    end if ;
    emit xpp

end loop
```

Table 3. Confirmation function

which forbids it: here we see two state variables computed in distinct locations. In a synchronous framework, both variables are computed at the same step. On the contrary, when each location has its own clock, if there is a dependency between these variables, the resulting state may depend on the order into which these variables are computed.

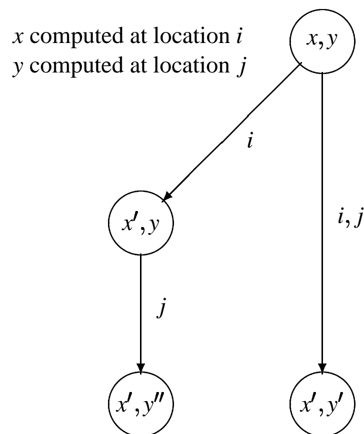


Fig. 12. A critical race

In this sense, sequential systems look very much like continuous unstable systems and cannot be directly implemented in a robust way. There are several ways for checking and enhancing the robustness of sequential systems:

- First, if state variables computed on distinct locations are independent, the critical race phenomenon does not take place and we are left to the combinational situation.
- Another interesting situation appears when dependent state variables cannot vary at the same step. Here also, there is no critical race. Now, there can be two reasons why state variables cannot vary at the same time:

  Timing reasons. If it is not the case, designers can add delays in order to robustify their programs.

  Causality reasons. For instance, in the mutual exclusion example of figure 4, we can transform the non robust program into the robust one by deciding that:

  y cannot raise but when z has gone down and conversely

  When this is used in conjunction with the mutex requirement stating that y and z cannot raise at the same time, we clearly forbid y and z to change at the same time, thus forbidding any race.
  Inserting causality chains disallowing races is thus a way of robustifying programs. This kind of programming is reminiscent of typical asynchronous programming methods like so-called *Message Sequence Charts* [18].

4.4 Mixed Systems

Now, more and more systems are mixed ones, and there is not clear way of dealing with them. Figure 13 illustrates the situation where a boolean signal is generated which changes when some continuous signal crosses a threshold. In our framework, where errors are associated with continuous signals, we can see that some error on the continuous signal induces some delay on the corresponding boolean. This fits quite nicely in our *error/delay* framework but for the fact that the relation between errors

and delays is non-linear: if the derivative at the crossing point vanishes, the delay can get unbounded!

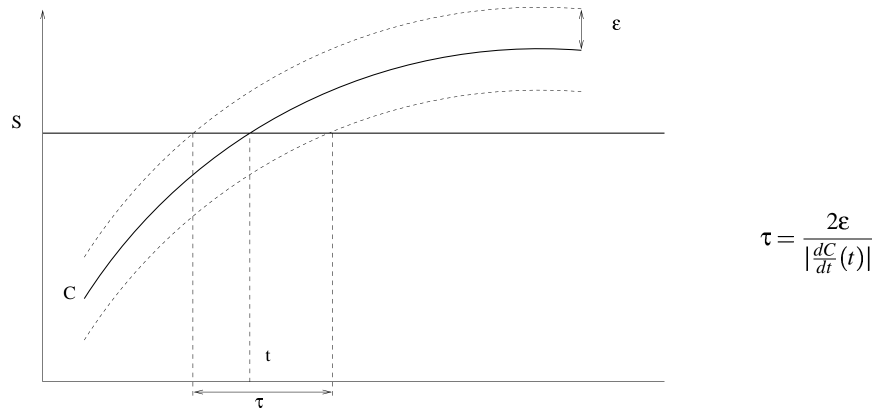


Fig. 13. Threshold crossing

5 Conclusion

In this paper we attempted to illustrate the evolution of ideas in control system programming. Starting from the early real-time languages and operating system propositions, we showed that control engineers elaborated their own practices which where generalised by computer scientists and gave birth to the synchronous paradigm.

But we also showed that for the sake of robustness and in particular distribution, some asynchrony had to be incorporated in the synchronous approach and we briefly reviewed the methods and theories used in practice for that purpose. What we can see here is that these methods and theories do not constitute, at present, a very mature and well established framework, and many efforts are still required in order to strengthen it and equip it with CAD tools.

Among the pending questions, we can cite:

– How could we merge together synchronous and asynchronous paradigms in order to obtain *robust by construction* distributed systems, while keeping the advantages of synchronous programming ?

– How can we encompass timing and causality within a coherent theory of robust discrete systems.
– How can we unify the continuous and discrete theories in order to obtain a satisfactory theory of mixed (hybrid) systems ?

Acknowledgments. The author is indebted to his colleagues of the Crisys project, mainly R. Salem from Verimag, M. Yeddes and R. David from Grenoble's Automatic Control Laboratory (LAG), C. Bodennec, C. Mazuet and N. Raynaud from Schneider Electric, R. Budde and A. Poigné from GMD, R. Mercadié from EADS-Airbus, for their contributions to the project and to the ideas proposed here. H. Kopetz, as a reviewer of the project, played a very important part in it by his always stimulating questions and many passionate discussions with O. Maler, E. Asarine, S. Tripakis from Verimag, J. Pulou from FranceTelecom and, last but not least, A. Benveniste from INRIA/IRISA, also contributed to the paper.

References

1. C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *Proc. CESA'96*, Lille, July 1996. 5
2. A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991. 5
3. J.L. Bergerand and E. Pilaud. SAGA; a software development environment for dependability in automatic control. In *SAFECOMP'88*. Pergamon Press, 1988. 5
4. G. Berry and G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 5, 7
5. G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process algebras and systems of communicating processes. In *Automatic Verification For Finite States Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer Verlag, 1990. 6
6. D. Brière, D. Ribot, D. Pilaud, and J.L. Camus. Methods and specification tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, December 1994. ERA Technology. 5
7. J. A. Brzozowski and C-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995. 15
8. P. Caspi and R. Salem. Threshold and bounded-delay voting in critical control systems. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 68–81, September 2000. 14
9. A. Chatha. Fieldbus: The foundation for field control systems. *Control Engineering*, pages 47–50, May 1994. 10
10. 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. 5
11. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, september 1992. 6

12. D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987. 5
13. C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–676, 1978. 3
14. K.J.Åström and B.Wittenmark. *Computer Controlled Systems*. Prentice-Hall, 1984. 5
15. H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: the MARS approach. *IEEE Micro*, 9(1):25–40, 1989. 11
16. G. LeGoff. Using synchronous languages for interlocking. In *First International Conference on Computer Application in Transportation Systems*, 1996. 5, 11
17. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proc. of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*. Springer Verlag, August 1992. 5
18. S. Mauw. The formalization of message sequence charts. *Computer Networks and ISDN Systems*, 28:1643–1657, 1996. 17
19. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983. 6
20. P.Caspi. What can we learn from synchronous data-flow languages. In O.Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 255–258. Springer, 1997. invited conference. 1