VERIMAG
UNITE MIXTE DE RECHERCHE

Centre Equation
2 avenue de Vignate
38610 GIERES
Tel. +33 4 76 63 48 48
Fax +33 4 76 63 48 50

The Quasi-Synchronous Approach to Distributed Control Systems

Crisys draft

October 2000

Centre National de la Recherche Scientifique | Universite Joseph Fourier | Institut National Polytechnique de Grenoble

Abstract

This report shows how the quasi-synchronous approach arose as a natural evolution going from networks of analog boards to distributed control systems on top of field busses. Primarily designed for continuous control, it progressively extends toward hybrid control and then raises synchronization problems, the more so as safety critical systems are considered. We propose here design, validation and fault tolerance methods adapted to this approach.

Contents

Chapter 1

Introduction

It seems that, from the very early times of SIFT and FTMP [24, 16], it was assumed that highly fault-tolerant control systems had to be based on exact voting and clock synchronization. This led to the discovery of consensus problems and Byzantine faults [22], and produced an important academic activity [11], culminating in the time-triggered approach to the development of these systems [17].

However, it also seems that, in practice, at least in the domain of critical control systems, the use of clock synchronization is not so frequent [6, 4, 14]. We believe there are historical reasons for this fact, which can be found in the evolution of these systems: control systems formerly used to be implemented with analog techniques, that is to say without any clock at all. Then, these implementations smoothly evolved toward discrete digital ones, and then toward computing ones. When a computer replaced an analog board, it was usually based on periodic sampling according to a real-time clock, and, for the sake of modularity, when several computers replaced analog boards, each one came with its own real-time clock.

Yet this raises the question of the techniques used in these systems for simulating, validating and implementing fault-tolerance without clock synchronization, and of the well-foundedness of these techniques which equip, up to now, some of the most safety critical control systems ever designed (civil aircrafts [6], nuclear plants [4], etc.)

This report aims at providing some answers to this question. It is organized as follows:

In chapter 2, we shall look at the architectural evolution from analog boards to distributed control systems. We show there how this evolution still preserved some qualities of autonomy between computing locations. Then, in chapter 3 we show how synchronous design tools allow a global system description, simulation and validation. However, these descriptions are non deterministic and yield problems of state explosion. We thus look, in chapter 4 for synchronous abstractions of these systems. First, we consider continuous control. Abstraction, here, is based on accuracy estimates. Then we look at discontinuous functions and take boolean ones as an illustration. Here, combinational functions appear as the analog of continuous ones. Yet boolean calculations are perfectly accurate but the analogy is based on a space-time trade-off. Thanks to confirmation functions, delays propagate from inputs to outputs in the same way as errors do for continuous computations. This technique can be extend toward some "robust" classes of sequential functions. Finally we propose a synchronization algorithm for handling non-robust sequential functions, which, in some sense, preserves the "qualities" of the quasi-synchronous architecture.

In the next chapter 5, we try to draw the practical consequences of the previous one. We first show that a control system validation (simulation, test, formal verification etc.), even in the centralized case, still requires a robustness analysis about the relations between the controller and its environment. Then we extend these requirements to the distributed case. A special section is devoted to the concurrent case, where programs are distributed as processes within the same computer.

Finally, in chapter 6, we consider fault tolerance. For continuous control, this leads us to threshold voting: two signals are considered to disagree if they differ for more than the maximum normal error. Similarly, combinational computations yield bounded-delay voting: two signals are considered to disagree if they remain different for more than the maximum normal delay. Extending this scheme to apply to sequential functions by transforming them into combinational ones, that is to say by bounded-delay voting on the state, leads to problems of Byzantine faults. However, we show here that 2/2 voting schemes are not sensitive to Byzantine faults and behave properly. This provides us with self-checking schemes which can then be used to build fault tolerant strategies by means of selective redundancy.

Chapter 2

The Architectural Evolution

Aircraft control systems illustrate this evolution which can also be found in many other fields of industrial control

2.1 Analog/Digital Communication

Starting from networks of analog boards, progressively some boards were replaced by discrete digital boards, and then by computers. Communication between the digital parts and the parts which remained analog was mainly based on periodic sampling (analog to digital conversion) and holding (digital to analog conversion), sampling periods being adapted to the frequency properties of the signals that traveled through the network. This allowed several technologies to smoothly cooperate. Figure 2.1 illustrates this evolution.

2.2 Serial Links

This technique was suitable up to the time when two connected analog boards were replaced by digital ones. Then these two also had to communicate and serial port communication appeared as the simplest way of replacing analog to digital and digital to analog communication as both can be seen as latches or memories. Figure 2.2 shows a typical situation borrowed from an automatic subway application. Each computer monitors a rail track section and runs a periodic program. Computers are linked together by serial lines

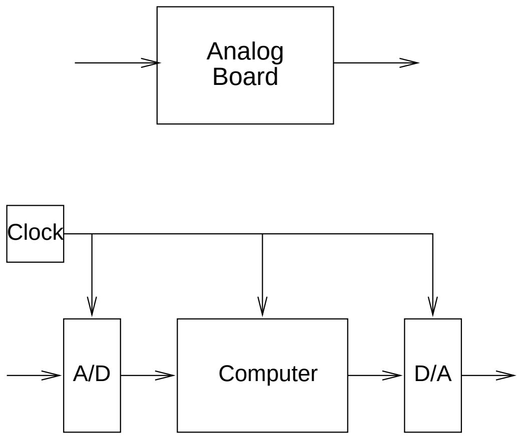


Figure 2.1: From analog to digital boards

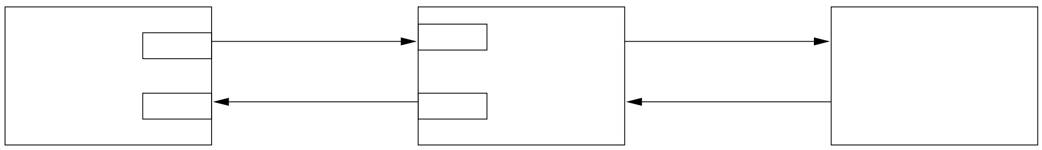


Figure 2.2: A point to point network

according to the topology of the track. Thus trains passing from a track section to another one are followed by the computers.

2.3 Field Busses

Then for the sake of optimization, these serial links are replaced by busses of several standards, (aircraft, industrial [10], automotive). Most of them, like Arinc429, just "pack" together several serial links, thus providing a kind of "shared memory" service, on top of which synchronization services can be implemented on need. Figure 2.3 illustrates this evolution, from networks of analog boards to local area networks.

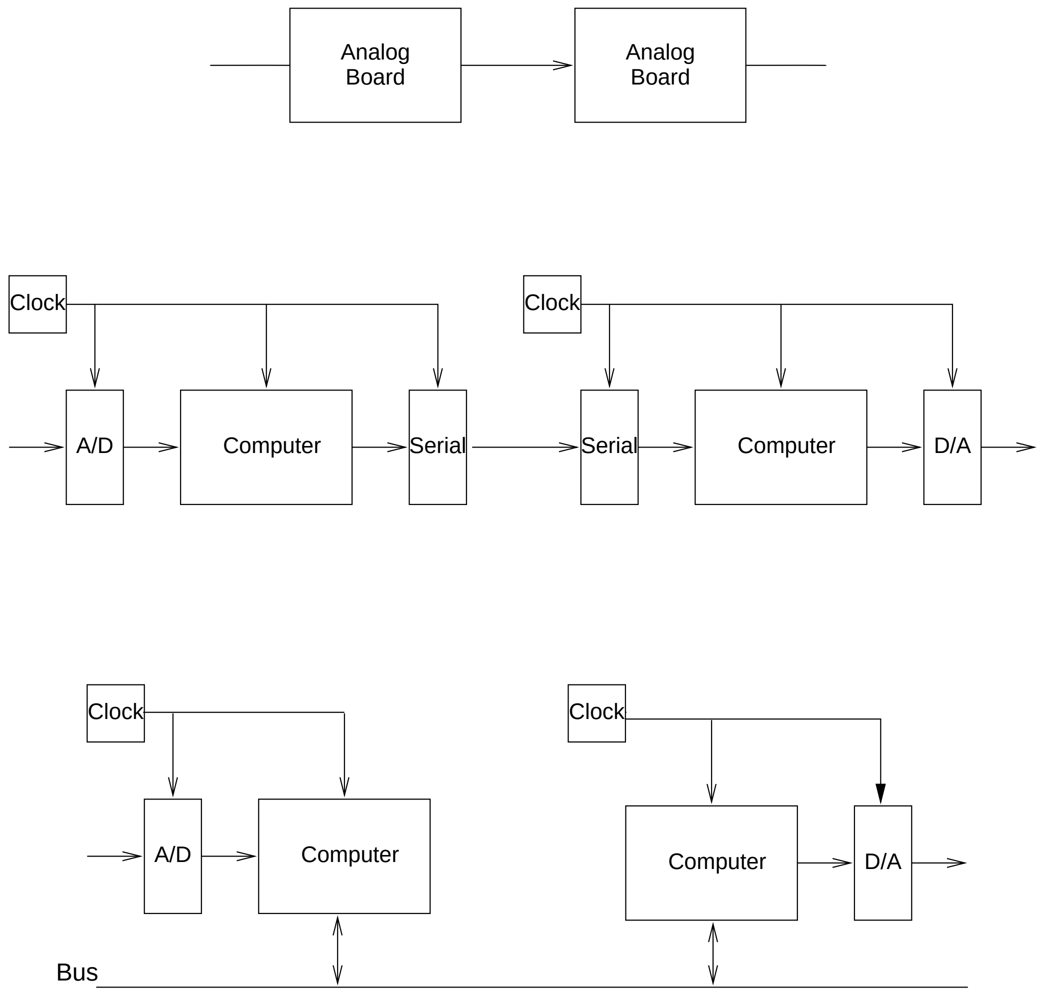


Figure 2.3: From analog networks to local area networks

2.4 Supervision

In most cases, this architecture is being added a supervisor, for monitoring purposes. The communication between the supervisor and field computers is however very different from the communication between field computers. It is an event-based communication which is assumed not to be time nor safety critical and which takes place either on special time slots of the field bus, or on a dedicated communication medium. The important fact, here, is that it should not perturbate neither the periodic behavior of field computers, nor their communication.

2.5 Provision against Byzantine Problems

In these very critical systems, Byzantine faults cannot be neglected and this is why some architectural precautions have to be taken in order to alleviate their consequences. For instance, these busses provide some protection against Byzantine problems [22], in the sense that they are based on broadcast: communication with several partners only involve one emission. Thus a failed unit cannot diversely lie to its partners. Then messages are protected by either error correcting and/or detecting codes which can be assumed to be powerful enough so that their failing be negligible with respect to the probabilistic fault tolerance requirements of the system under consideration.

2.6 Communication Abstraction

According to what precedes, we can quite precisely state an abstract property of this kind of communication medium, which is a bounded delay communication property:

Property 1. *First, we assume that every process P is periodic with a period varying between small margins:*

$$T_{Pm} \leq T_P \leq T_{PM}$$

Then,

Property 2. *Let T_{sM} and T_{rM} be the respective maximal periods of the sender and of the receiver, and n the maximum number of non negligible consecutive failed receives (in the case of error correction, $n = 1$).*

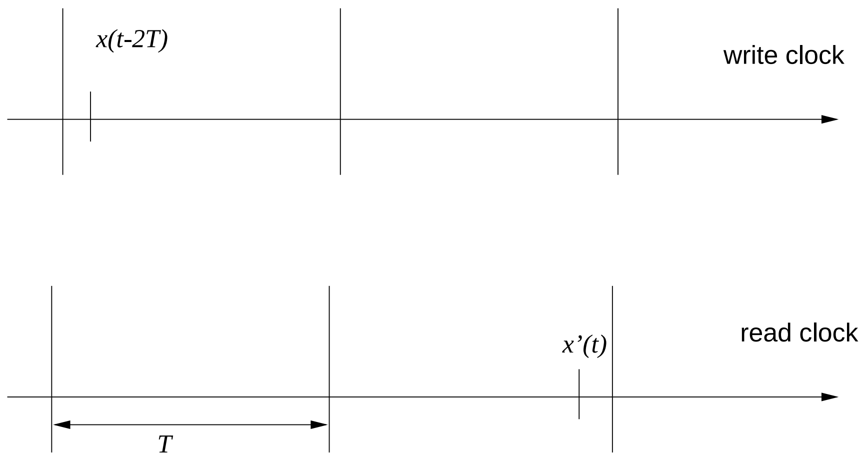


Figure 2.4: Worst delay situation

Then the value $x_r(t)$ known at any time t by the receiver of some signal x_s communicated by the sender is some $x_s(t')$, where

$$|t - t'| \leq T_{rM} + nT_{sM}$$

Figure 2.4 illustrates this situation in the case $n = 1$ and $T_s = T_r$. The worst delay happens when reads occur just before writes.

Definition 1. *A signal x' is a τ* **bounded delay image** *of a signal x if there exists a monotonic (retiming) function $t' : R^+ \to R^+$ such that*

$$\forall t \in R^+, 0 \leq t - t'(t) \leq \tau \text{ and } x'(t) = x(t'(t))$$

2.7 Summary

What seems to characterize this evolution is an effort toward keeping the autonomy of computing agents, which can be seen as a prerequisite of robustness, on top of which fault tolerance strategies can be built. It is not incidental that, in the aircraft industry, this architecture is called the "federated computer architecture". Three main features seem to contribute to this autonomy:

- Each computer is a complete one, including its own clock and even possibly its own power supply.

- Communication between computers is non-blocking, based on periodic reads and writes, akin to periodic sampling.
- Fault tolerance is main based on self-checking: 2/2 votes allow self-checking dual modules to exhibit fail-silent behaviors, on top of which more elaborated fault tolerance strategies can be implemented.

In the sequel, we try to understand how does this architecture works and how applications can be safely implemented on top of it.

Chapter 3

A Synchronous Tool set for Quasi-Synchronous Systems

In this chapter we show how synchronous design tools allow a global system description, simulation and validation. We first describe our notation. Then we show how to describe systems implemented on a quasi-synchronous architecture and how to simulate them.

3.1 Synchronous Data-Flow Notations

In the sequel, algorithms are expressed using a functional notation, that is to say by abstracting over time indices, in order to stay consistent with design tools. Thus, a signal definition:

$x_1 = x_2$ means $\forall n \in N : x_1(nT) = x_2(nT)$.

3.1.1 Usual Operators

An operation:

$(x_1 - x_2)(nT)$ means $x_1(nT) - x_2(nT)$

and similarly,

(`if` c `then` x `else` y)(nT) means `if` $c(nT)$ `then` $x(nT)$ `else` $y(nT)$

Note yet that the "=" sign is overloaded: it is both used as expressing a signal definition as above, and as the boolean valued binary operator yielding true if the two arguments are equal and false otherwise:

$(x_1 = x_2)(nT)$ means $x_1(nT) = x_2(nT)$

Hopefully, the context should allow a clear distinction between the two cases.

3.1.2 Time Operators

Three special time operators are provided as in Lustre [12] and Scade:

Unit Delay

`fby` is the unit delay for periodically sampled signals[1]:

$$x0 \texttt{ fby } x(nT) = \begin{array}{l} \texttt{if } n = 0 \\ \texttt{then } x0 \\ \texttt{else } x((n-1)T) \end{array}$$

Provided this definition, dynamical systems can be expressed and programmed. Here, a dynamical system $S(v, u)$ is represented by its transition function F.

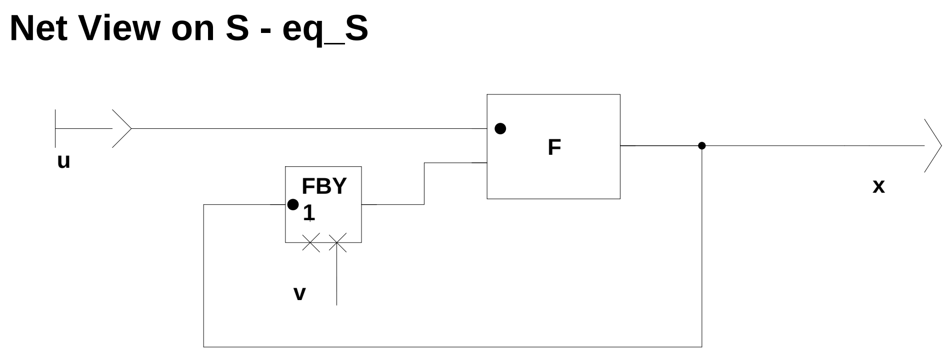
$$\begin{array}{rcl} S(v,u) & = & x \\ \texttt{where } x & = & F(v \texttt{ fby } x, u) \end{array}$$

Figure 3.1 shows the corresponding Scade diagram and Lustre program.

Sampling

This formalization does not allow us to account for systems running on a slower clock than the "basic"sampling clock. This can be achieved thanks to

[1]Indicating the sampling period allows mixing both time continuous and sampled systems within the same equational framework; this feature is currently provided in design tools such as Simulink and Matrix_X.




```
const v:type
node S (u:type) returns (x:type);
let
    x = F(v->pre x,u);
tel
```

Figure 3.1: A dynamic system block diagram and program

u	u_0	u_1	u_2	u_3	$\ldots$
v `fby` u	v	u_0	u_1	u_2	$\ldots$
c	f	t	f	t	$\ldots$
u `when` c		u_1		u_3	$\ldots$
u'		u'_0		u'_1	$\ldots$
`current`(v, c) u'	v	u'_0	u'_0	u'_1	$\ldots$

Table 3.1: Data-flow primitives

the `when` sampling function whose behavior shown at table 3.1. Intuitively, it erases from the u sequence the elements corresponding to "false" elements of the c sequence.

Holding

The converse holding function `current`$(,)$ (derived from the Lustre `current` function) is shown at table 3.1. Intuitively, it operates by filling the holes of the u sequence (signaled by "false" values of c) with the value of u the last time c was "true". If none, an initial value v is provided.

Activation Conditions

Scade activation conditions correspond to a combined use of sampling and holding. Signals are sampled at the input of a system and conversely hold at the output. Here also default values are to be provided.

3.2 Communication Abstraction

Let us show here that, thanks to these building blocks, we can faithfully retrieve, as a theorem, the properties of the quasi-synchronous communication protocol stated at 2.6. This will be done by first formalizing the communication medium and then by stating an abstract property of unsynchronized periodic clocks sharing approximately the same period.

3.2.1 Shared Memory

Given a sequence u written in the shared memory at clock cw and an initial content v, the current content of the memory can be expressed as:

$$mem(v, cw, u) = v \ \texttt{fby} \ (\texttt{current}(v, cw) \ u)$$

where the delay accounts for short[2] undetermined transmission delays.

Then, the sequence read at clock cr is :

$$u' = mem(v, cw, u) \ \texttt{when} \ cr$$

3.2.2 Formalizing Periodic Clocks

This could be done in some real-time framework, such as timed automata [2], but, for the sake of simplicity, we prefer here to characterize the fact that two independent clocks have approximately the same period by saying that:

Any of the two clocks cannot take the value "t"more than twice between two successive "t" values of the other one.

This can be formalized by saying that the boolean vector stream composed of the two clocks should never contain the subsequence:

$$\begin{bmatrix} t \\ - \end{bmatrix} \cdot \begin{bmatrix} f \\ f \end{bmatrix}^* \cdot \begin{bmatrix} t \\ f \end{bmatrix} \cdot \begin{bmatrix} f \\ f \end{bmatrix}^* \cdot \begin{bmatrix} t \\ - \end{bmatrix}$$

nor the one obtained by exchanging coordinates. (Here, $-$ is a wild card representing any of the two values $\{t, f\}$.)

Now, such regular expressions yield finite state recognizability and can be associated a finite-state recognizing dynamic system $Same\_Period_2$[3].

Furthermore, replacing in what precedes 2 by n allows defining similar $Same\_Period_n$ systems.

---

[2] Significantly shorter than the periods of read and write clocks. If longer transmission delays are needed, modeling should be more complex.

[3] This can be automatically generated in Lustre, thanks to the Reglo tool [23].

3.2.3 Proving the Abstract Communication Property

Intuitively, if read and write clocks have the same period, the read value should not be "too far" from the written one. This can be formalized as the following theorem:

Theorem 3.2.1. *If $Same\_Period_2(cw, cr)$ always holds, then the following property always holds:*

$$\begin{pmatrix} & mem(v, cr, u') = mem(v.cw, u) \\ \vee & mem(v, cr, u') = mem(v, cw, v \texttt{ fby } u) \\ \vee & mem(v, cr, u') = mem(v, cw, v \texttt{ fby } v \texttt{ fby } u) \end{pmatrix}$$

where $u' = mem(v, cw, u)$ `when` cr

Proof This "theorem" is automatically proved by the Lustre prover Lesar [13].

It means that any used internal value must have been produced within a two period time interval. This is exactly what is stated in property 2 when taking $T_r = T_s$ and $n = 1$.

3.3 Quasi-Synchronous Programs

We are thus in a position to faithfully represent, simulate, test generate and even formally prove quasi-synchronous programs. For instance, a faithful Scade representation of the architecture shown at figure 2.2 is displayed at figure 3.2

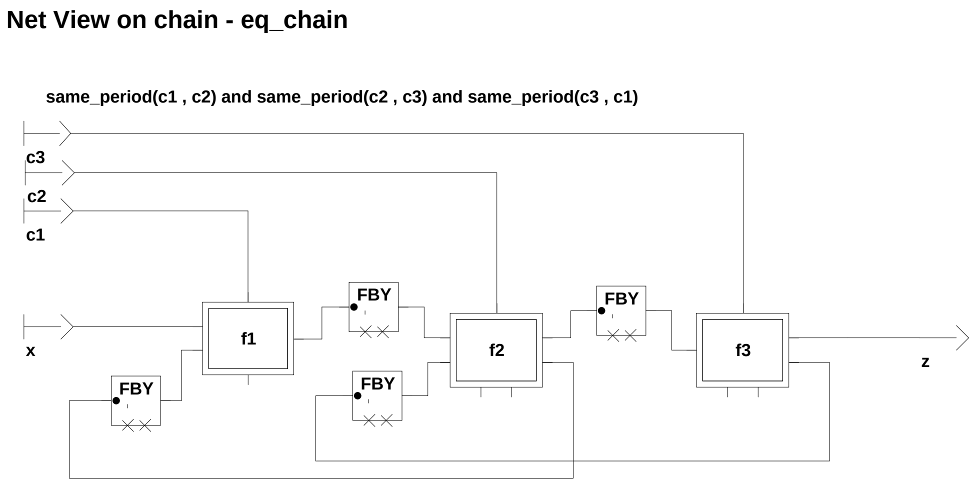


Figure 3.2: Scade diagram corresponding to the system of figure 2.2.

Chapter 4

Synchronous Abstraction

The descriptions proposed in the previous chapter are non deterministic. Using them for formal verification and even for simulation and test generation will frequently lead to problems of state explosion. We thus look for synchronous abstractions of these systems, in which we can "forget" the multiple independent clocks and verify them as if they were perfectly synchronous. First, we consider continuous control. Abstraction, here, is based on accuracy estimates. Then we look at discontinuous functions and take boolean ones as an illustration. Here, combinational functions appear as the analog of continuous ones. Yet boolean calculations are perfectly accurate but the analogy is based on a space-time trade-off. Thanks to confirmation functions, delays propagate from inputs to outputs in the same way as errors do for continuous computations. This technique can be extend toward some "robust" classes of sequential functions. Finally, we address the case of non robust sequential systems which then require some kind of logical synchronization. We propose here a synchronization algorithm which retains most of the appealing features of the quasi-synchronous approach.

4.1 Continuous Control

4.1.1 Continuous Signals and Functions

Most basic accuracy computations on continuous signals and functions over signals can be based on standard uniform continuity:

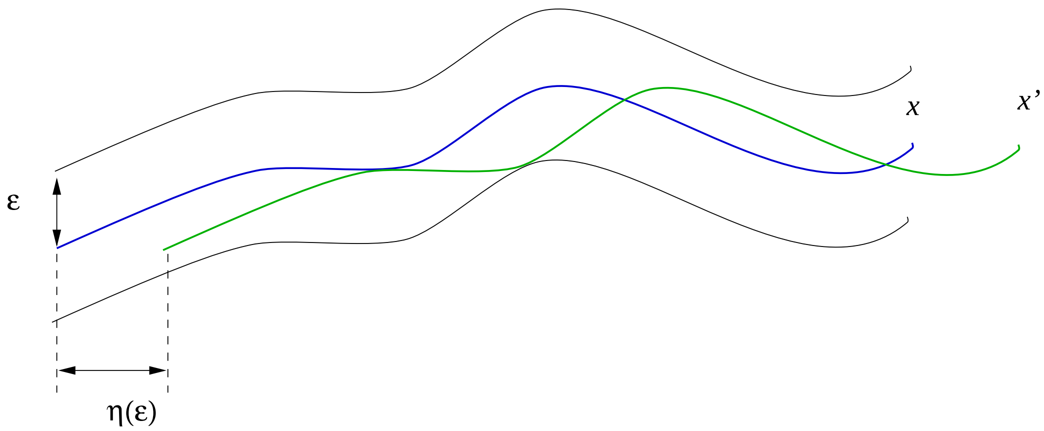


Figure 4.1: An uniformly continuous signal

Definition 2. *A function $f \in R^n \rightarrow R^m$ is uniformly continuous if there exists an error function $\eta_f \in R^+ \rightarrow R^+$ such that, for all $x, x' \in R^n$ and $\epsilon \in R^+$:*

$$||x' - x|| \leq \eta_f(\epsilon) \Rightarrow ||f(x') - f(x)|| \leq \epsilon$$

Figure 4.1 illustrates the case of an uniformly continuous signal: delaying the signal by $\eta(\epsilon)$ keeps the values within an ϵ tube.

As function composition preserves uniform continuity, this easily allows the computation of bounds for systems made of uniformly continuous static functions fed by uniformly continuous signals through bounded delay networks. This, in turn, allows the computation of voting thresholds or more precisely the computation of periods such as to reach some accuracy or some voting threshold. For instance,

$$|t' - t| \leq T = \eta_x(\eta_f(\epsilon)) \Rightarrow ||f(x(t')) - f(x(t))|| \leq \epsilon$$

4.1.2 Dynamical Systems

However, static functions are quite rare in control algorithms, and one would rather find dynamical functions. There can be at least three different cases:

The stable case consists of seeing dynamic systems as uniformly continuous functions over normed spaces of time signals. If we want to compute

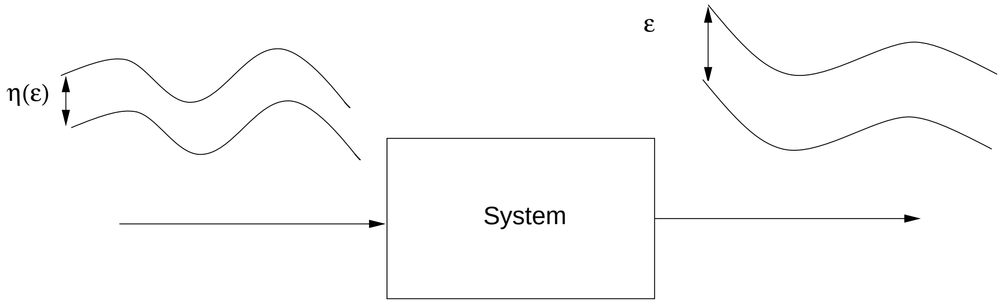


Figure 4.2: An uniformly continuous system

deterministic voting thresholds, an adequate norm is the $\mathcal{L}_\infty$ one:

$$||x' - x||_\infty = sup\{||x'(t) - x(t)|| \mid t \in R^+\}$$

Figure 4.2 illustrates the case of an uniformly continuous system: if the input signal stays within a $\eta(\epsilon)$ tube, the output stays within an ϵ tube.

If f is an uniformly continuous dynamical function for this norm, fed with uniformly continuous signals, we can easily reach the same bound as for static ones:

Theorem 4.1.1. *If x' is a τ bounded delay image of x, if x is uniformly continuous with error function η_x, and if f is uniformly continuous with error function η_f, then*

$$\tau \leq \eta_x(\eta_f(\epsilon)) \Rightarrow ||f(x') - f(x)||_\infty \leq \epsilon$$

Proof:

- $||x' - x||_\infty \leq \eta_f(\epsilon) \Rightarrow ||f(x') - f(x)||_\infty \leq \epsilon$ from the uniform continuity of f
- $\tau \leq \eta_x(\eta_f(\epsilon)) \Rightarrow ||x' - x||_\infty \leq \eta_f(\epsilon)$ from the fact that the delay between x' and x is less than or equal to τ and x is uniformly continuous.

In some sense, this uniform continuity is closely linked to stability: this could be rephrased by saying that stable dynamical functions behave like static ones.

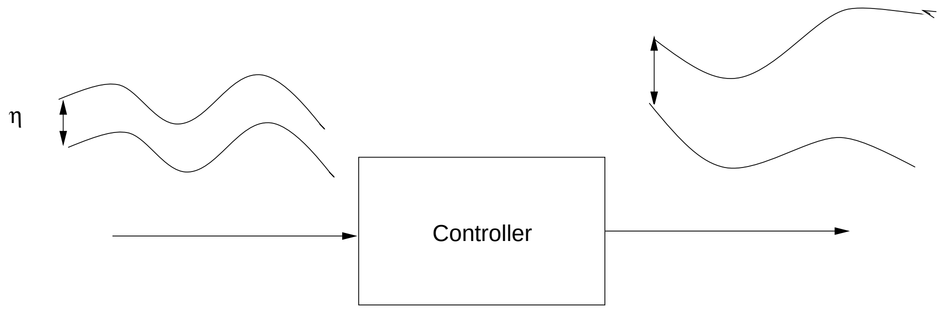


Figure 4.3: An unstable controller

The Stabilized Case Unfortunately, controllers are often not stable: even a mere PID has an integration stage which makes it asymptotically unstable and, hence, not uniformly continuous. Thus errors cannot be monitored. Figure 4.3 illustrates this situation. However, the picture changes if we consider this controller in closed-loop with the system it controls. A typical situation is shown at figure 4.4. This closed-loop system "computes" the following system of functional equations:

$$\begin{aligned} X &= \texttt{Controller}(U, Y0 \ \texttt{fby} \ Y) \\ Y &= \texttt{Plant}(X, Z) \end{aligned}$$

where `Control` is the controller , `Env` is the system under control, U is the vector of set values, X holds the control signals, Y contains the measurements from the process to be controlled that are fed back to the controller and Z contains the external perturbations that the system under control is subjected to. In order to yield an overall stable system, the system under control often requires a controller that, when viewed in isolation of the control system, is not stable. But the overall stable system still computes X as an uniformly continuous function of signals U and Z.[1] This allows bounds and thresholds to be computed as before.

The unstable case However, even in the case of overall stable systems, it may be that some unstable variables are computed by the controller.

[1]This phenomenon is quite important and contradicts many wishful thoughts about control system implementation such as composability and separation of concern between automatic control and computer science people.

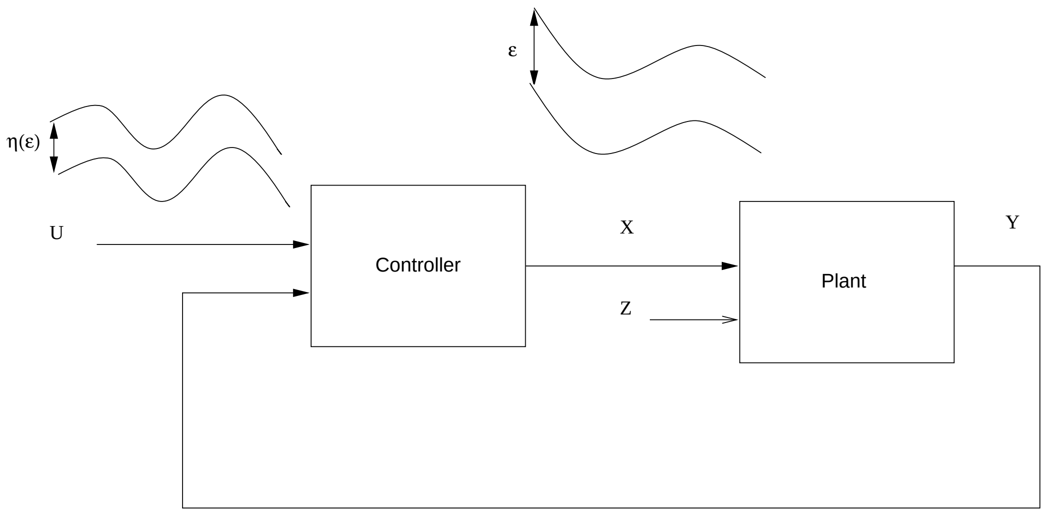


Figure 4.4: A closed-loop control system

Here, clock synchronization can be useful. But, in several cases, computing unstable values is a problem, regardless of fault tolerance, and some re-synchronization is algorithmically provided, so as to get stable computations, for instance thanks to Kalman filtering techniques. An example of such a situation can be found in [3].

4.2 Combinational Boolean Functions

The case of combinational boolean functions closely looks like the static continuous ones, but for the fact that no error function is available. We thus need to elaborate some notions which help in recovering it.

4.2.1 Uniform Bounded Variability

Periodically sampling boolean signals is only accurate if those signals do not vary too fast with respect to the sampling period. Bounded variability (closely linked to non Zenoness [1]) is intended to capture this fact. However, it is not strong enough a property in the same way as continuity is not strong enough for computing error bounds. What is needed in fact is the uniform

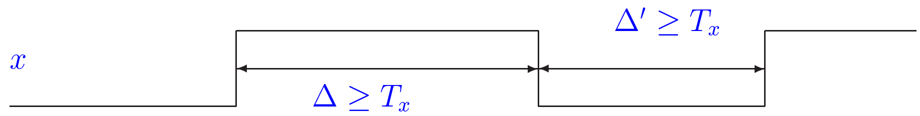


Figure 4.5: An uniform bounded variability signal

version of it, a possible definition, which only presents a small deviation from the one in [9], being as follows:

Definition 3. *A signal $x \in R^+ \to B^n$ has uniform bounded variability if there exists a discontinuity count function $c_x \in R^+ \to N$ such that any time interval of duration not larger than τ contains a number of discontinuities not larger than $c_x(\tau)$:*

$$\forall t_1, t_2 : \; t_2 - t_1 \leq \tau \;\Rightarrow\; |\{t \mid t \in [t_1, t_2] \;\wedge\; x(t^-) \neq x(t^+)\}| \leq c_x(\tau)$$

where $x(t^-)$ and $x(t^+)$ represent respective left and right limits at time t. Thus, any time t such that $x(t^-) \neq x(t^+)$ is a discontinuity point.

This framework then allows us to define which boolean signals and tuples can be sampled without loosing too much information.

Definition 4. *A boolean signal x can be sampled with period T if $c_x(T) = 1$.*

This ensures that no value change will be lost by sampling.

Another way, even more practical of defining the same thing is to define the minimum stable time interval:

Definition 5. *The minimum stable time interval T_x of a boolean signal x is the largest time interval such that $c_x(T_x) = 1$:*

$$T_x = sup\{T \;|c_x(T) \leq 1\}$$

Thus the sampling period should be smaller than T_x. Figure 4.5 illustrates this situation.

We can now relate bounded delays and minimum stable time interval:

Theorem 4.2.1 (Bounded delay and minimum stable time interval). *If x' is a τ bounded delay image of x and if T_x is the minimum stable time interval of x, then the minimum stable time interval of x' is:*

$$T_{x'} = T_x - \tau$$

provided $\tau < T_x$.

The proof comes from the worst case: x changes once and takes a maximum delay, and then changes again with 0 delay.

Remark: this result can be improved by considering both minimum τ_m and maximum τ_M delays: then

$$T_{x'} = T_x - (\tau_M - \tau_m)$$

Yet the extension to independent tuples is more difficult, since we cannot ensure that no value change will be lost. This is a general problem which arises in any periodically sampled system. In most cases, the best we can do is to choose the least period such that each component of the tuple is well sampled in the preceding sense. Nevertheless, we can relate the least minimum stable time interval T_x of each component of a n-tuple X to some time interval where the value of the tuple remains constant:

Theorem 4.2.2. *Let X be an n-tuple such that each component has a minimum stable interval larger that T_x. Then, in each time interval of duration larger than T_x, there exists a time interval of duration at least $\frac{T_x}{n+1}$ where the value of the tuple remains constant.*

The proof is based on the fact that in a T_x time interval any component of the tuple cannot change more than once. In the worst case each component changes once, and this partitions the interval into $n+1$ sub-intervals. Now it cannot be the case that every sub-interval is smaller than $\frac{T_x}{n+1}$ because their sum would be smaller than T_x.

4.2.2 Bounded Variability and Bounded Delays: Confirmation

However, delays do not combine nicely as errors do: the effect of errors on the arguments of a computation amounts to some error on the result. This

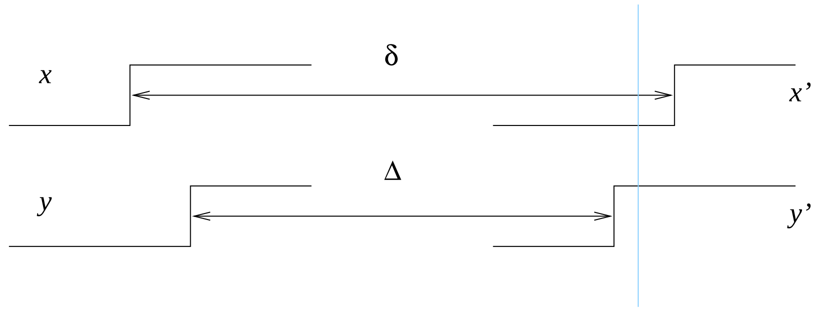


Figure 4.6: Tuples of delayed signals may not be delayed tuples

is in general not true for delays: if the arguments of a computation come with distinct delays, the result may not always be a delayed image of the ideal (not delayed) result. Figure 4.6 shows such a situation. Fortunately, some filtering procedures make it possible to change incoherent delays into coherent ones: this is known as confirmation procedures:

Assume a tuple $X' = \{x'_i, i = 1, n\}$ of boolean signals coming in one unit of period T from different units such that it is the image of an ideal tuple X through bounded delays:

$$\forall t, i = 1, n, \exists \tau_{t,i}, 0 \leq \tau_{t,i} \leq \tau_X : x'_i(t) = x_i(t - \tau_{t,i})$$

where we assume that all delays have the same upper bound τ_X.

We consider the following confirmation function:

Definition 6 (Confirmation function).

$$\begin{array}{lcl}
\mathit{confirm}(X', nmax) & = & X'' \\
\\
\texttt{where}\ X'',\ n & = & \texttt{if}\ X' \neq X0\ \texttt{fby}\ X' \\
 & & \texttt{then}\ X0\ \texttt{fby}\ X'',\ \ 0 \\
 & & \texttt{else}\ \ \texttt{if}\ 0\ \texttt{fby}\ n < nmax - 1 \\
 & & \qquad\ \texttt{then}\ X0\ \texttt{fby}\ X'',\ \ 0\ \texttt{fby}\ n + 1 \\
 & & \qquad\ \texttt{else}\ X',\ \ 0\ \texttt{fby}\ n
\end{array}$$

- this confirm function stores a counter n with initial value 0, and its previous output, with some known initial value $X0$,

- whenever its input changes , it outputs its previous output and resets the counter,
- else, if the counter has not reached $nmax - 1$, it increments it and outputs the previous output,
- else it outputs the current input and leaves the counter unchanged.

We further assume that $nmax$ is the maximum number of samples that can occur within the maximum delay:

$$nmax = E(\frac{\tau_X}{T_m}) + 1$$

where E denotes the integer part function, and that the minimum stable interval of each component of the tuple T_x exceeds $(n.nmax + 1)T_M$. We then can prove:

Theorem 4.2.3. *The output of the confirm function is a delayed image of the original tuple:*

$$\forall t, \exists \tau_t, 0 \leq \tau_t \leq (n.nmax + 1)T : \mathit{confirm}(X', nmax)(t) = X(t - \tau_t)$$

The proof consists of considering the worst case: one component changes just after an execution, and then each component of the tuple changes just before the confirmation time $nmaxT$ is reached. Figure 4.7 displays the corresponding Scade program.

This shows that incoherences due to variable delays can be changed into coherent delays. But we can note here an interesting by-product of this confirm function:

Corollary 1. *The output of a confirm function $confirm(X, nmax)$ remains constant for time intervals of duration at least $nmaxT_m$.*

4.3 Robust Sequential Functions

Robustness for sequential functions is an important topic, which encompasses many different disciplines ranging from distributed programming to asynchronous circuit theory [8]. An ultimate way of achieving it consists of programming systems in an asynchronous framework, for instance using ADA

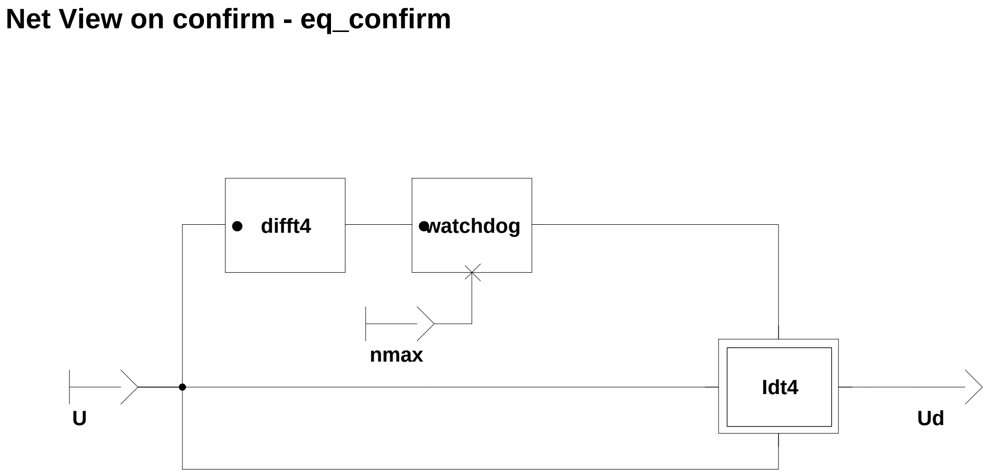


Figure 4.7: Confirmation function

tasking, Petri nets, SDL [21], etc. In doing it, we know that we operate within a semantic which only allows robust programs to be written, as this semantics is independent of the relative speeds of the component programs. In this setting, the combination of tools, like for instance Scade and SDL, presents a lot of interest.

However, this somewhat contradicts the fact that designers of the domain are more familiar with synchronous languages which better match their cultural background (differential and difference equations, synchronous state machines, etc.). A solution to the problem of designing robust sequential functions using these languages could consist of designing specific asynchronous libraries within these languages. We already know from the pioneering work of Milner [20] that this is possible.

Another solution, may be more practical, consists of being confident in the skill of the designer, and just provide tools for checking that the designed systems are robust enough. This is yet a difficult task, which is akin to reverse engineering.

In this section we address two topics. In the first one, we try to characterize a subset of sequential functions which behave, with respect to distribution, "like" combinational ones. The second one discusses the problem of speed independence.

4.3.1 Quasi-Combinational Systems

We list and define here some properties of a sequential system whose combination allows it to look like a combinational one with respect to distribution.

For the sake of simplicity, we do not distinguish here between states and outputs. Hence, a sequential system will be defined by its transition function F, such that:

$$X = F(X0 \texttt{ fby } X, U)$$

Stability: it is likely indeed that distributed programs will have to run faster in order to produce behaviors comparable to those of centralized programs. But running a synchronous program faster on the same inputs will in general deeply modify its behavior. This is why we may expect it easier to distribute stable systems rather than unstable ones, stable systems being those that can run faster without too much changing their behaviors. More formally stability will be associated with transition functions of systems:

Definition 7 (Stability). *A transition function F is n_stable if*

$$F^{n+1} = F^n$$

where F^n is the nth iterated of F:

$$\begin{array}{lcl} F^0(x,u) & = & x \\ F^{n+1}(x,u) & = & F(F^n(x,u),u) \end{array}$$

This means that when the inputs do not change, the state of the dynamic system will stabilize.

Any n_stable function can easily be turned into a 1_stable function by taking:

$$F' = F^n$$

Order insensitivity: another feature of distributed systems is that iterations of the transition function are not computed in a parallel synchronous fashion, but in a sequential (chaotic ordered) way. Let us call "order-insensitive"[2] a transition function which can be executed in that way:

Definition 8 (Order-insensitivity). *A transition function F is order-insensitive if, for any initial state x, any input u, and any permutation σ of the interval $1, \dots n$,*

$$F(x,u) = y(n)$$

where y is the sequence:

$$\begin{array}{rcl} y(0) & = & x \\ y_i(p+1) & = & \left\{ \begin{array}{ll} F_i(y(p),u) & \textit{if } p+1 = \sigma(i) \\ y_i(p) & \textit{otherwise} \end{array} \right. \end{array}$$

Clearly, any transition function F can be turned into an order-insensitive one F'. The most general (and also the most expensive) way of achieving this goal consists of expanding the state space in such a way that each component of the state function computes the whole state space:

$$F'_i(x',u) = F(x'_i,u)$$

As a matter of fact, this technique achieves a stronger goal, which can be defined as *state decoupling*:

Definition 9 (Decoupling). *A transition function F is decoupled if and only if each component depends only of the corresponding state component. More formally, it exists a function F' such that for all i:*

$$F_i(x,u) = F'_i(x_i,u)$$

Confluence: another desirable property for distribution is confluence. It means that input changes can be arbitrarily composed while yielding the same final state. More formally,

Definition 10 (Confluence). *A 1_stable transition function F is confluent if for any state x, input u and input changes δ and δ' acting on distinct input variables (for all i, $\delta_i.\delta'_i = 0$):*

$$F(F(F(x,u), u+\delta), u+\delta+\delta') = F(F(x,u), u+\delta+\delta')$$

[2]By analogy with the "delay-insensitivity" of asynchronous hardware [7]

where + is the "exclusive-or" operation on booleans, implicitly extended to operate point-wise.

Confluent functions are easier to distribute than non confluent ones. Thus, checking for confluence can be an important issue when it comes to distribution. However, confluence is a very restrictive property and we cannot limit ourselves to distributing confluent functions.[3]

We may need to strengthen this definition by considering local confluence:

Definition 11 (Local confluence). *A 1_stable transition function F is locally confluent if any of its component functions is confluent when considering other state variables as inputs.*

Quite surprisingly, confluence even combined with order insensitivity does not imply local confluence. However this holds if we replace order insensitivity with decoupling.

We are now in a position to define what we call quasi-combinational distributed systems:

Definition 12 (Quasi-Combinational). *A collection of transition functions $F_i, i = 1, n$ is quasi-combinational if:*

- *the global system of transition function:*

  $$F(X, U) = (F_1(X, U), \ldots F_n(X, U))$$

  is 1_stable,

- *and either F is decoupled and confluent*

- *or F is order-insensitive and locally confluent.*

[3]In asynchronous circuit theory, it seems that both order-insensitivity and confluence are not distinguished. Both lead to races among internal variables. The usual assumption is the "fundamental mode" which assumes that two input variables cannot change at the same time and that an asynchronous circuit has enough time to stabilize before the next input change. Thanks to this assumption, the confluence problem does not occur. In the case of distributed systems, the fundamental mode has no longer any meaning, hence the confluence problem has to be considered. It is important to distinguish between the two properties because order-insensitivity is not an intrinsic property of a function (but only of its implementation) while confluence is.

4.3.2 Enforcing System Robustness

However, these quasi-combinational systems exhibit a very low degree of cooperation and we know many robust systems which do not fulfill the preceding criteria.

The rendez-vous example A typical example is the one of a "rendez-vous". Let us consider two locations and a special state value x_{rv} which allows the locations to agree on waiting for each other. Once a location has reached this state, it stays in it whatever be its inputs, until the other location also reaches the same state:

$$\begin{aligned}
f_1(x_{rv}, y, u) &= x_{rv} \\
f_1(x_{rv}, y, u') &= x_{rv} \\
f_1(x_{rv}, x_{rv}, u) &= x' \\
f_1(x', x_{rv}, u') &= x''
\end{aligned}$$

As we can see, the location 1 transition function is not even locally confluent.

Among the many possible robust systems, quasi-combinational ones enhanced with rendez-vous capabilities seem a good choice which has emerged from the long lasting efforts of computer scientists for defining robust distributed systems [15, 19]. However we do not clearly know, at this time,

- neither how to check that robustness,
- nor how to encompass it within synchronous languages, despite the efforts done in [5].

4.4 Non Robust Sequential Functions

Non robust sequential systems require some kind of logical synchronization. We propose here a synchronization algorithm which retains most of the appealing features of the quasi-synchronous approach.

4.4.1 A Synchronization Algorithm

The description of this algorithm is as follows:

1. Every unit i maintains a counter n_i. Initially these counters are set to 0. This corresponds to an idle state.
2. When $n_i = 0$, if unit i sees a proper input change and no input change from other units, it broadcasts its input and state value change and sets its counter to 3.
3. When $n_i = 0$, if unit i sees input changes from other units, it possibly broadcasts its input and state value changes and sets its counter to 2.
4. Then at each step, each unit decrements its counter down to 0
5. When $n_i = 1$, every unit i executes a transition.

This can be summarized by the following "pseudo-Scade" program of table 4.1:

4.4.2 Proof

The proof is very simple and based on timing properties of periodic processes:

- Let t be the time at which the first unit detects an input change,
- τ, τ' non zero processing and transmission delays,
- and T the period of every unit.

Then,

- $t + \tau + T + \tau$ is the time of the latest possible write on the bus, corresponding to a unit which executes just before the first write $(t + \tau)$ (hence it does not see it); then it executes again after T and needs τ to write on the bus,
- $t + \tau + 3T$ is the time at which this unit executes a transition; it is the latest possible transition execution time,
- $t + 3T$ is the time at which the unit which wrote first executes a transition,

$$
\begin{array}{rcl}
n_i & = & \texttt{if } 0 \texttt{ fby } n_i = 0 \wedge u_i \neq u_{i0} \texttt{ fby } u_i \wedge U'_i = U0 \texttt{ fby } U'_i \\
 & & \texttt{then } 3 \\
 & & \texttt{else } \; \texttt{if } 0 \texttt{ fby } n_i = 0 \wedge U'_i \neq U0 \texttt{ fby } U'_i \\
 & & \qquad \texttt{then } 2 \\
 & & \qquad \texttt{else } \; \texttt{if } 0 \texttt{ fby } n_i > 0 \\
 & & \qquad \qquad \texttt{then } 0 \texttt{ fby } n_i - 1, \\
 & & \qquad \qquad \texttt{else } 0 \texttt{ fby } n_i \\
u''_i, x''_i & = & \texttt{if } 0 \texttt{ fby } n_i = 0 \wedge u_i \neq u_{i0} \texttt{ fby } u_i \wedge U'_i = U0 \texttt{ fby } U'_i \\
 & & \texttt{then } u_i, x_i \\
 & & \texttt{else } \; \texttt{if } 0 \texttt{ fby } n_i = 0 \wedge U'_i \neq U0 \texttt{ fby } U'_i \\
 & & \qquad \texttt{then } u_i, x_i \\
 & & \qquad \texttt{else } u0_i \texttt{ fby } u''_i, x0_i \texttt{ fby } x''_i \\
x_i & = & \texttt{if } 0 \texttt{ fby } n_i = 1 \\
 & & \texttt{then } \mathcal{F}_i(X'_i, U'_i) \\
 & & \texttt{else } x0_i \texttt{ fby } x_i
\end{array}
$$

Table 4.1: The Synchronization Algorithm: u_i and x_i the proper input and state of unit i, U'_i and X'_i are the total input and state vectors as seen on the network by unit i, u''_i and x''_i are the input value and state broadcast by unit i.

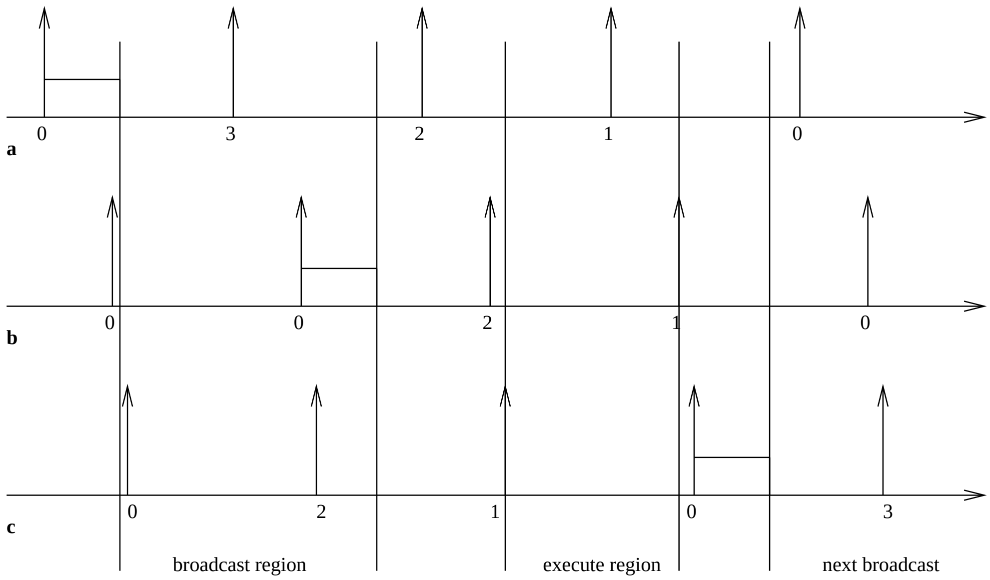


Figure 4.8: Proof illustration: a) is a "first to write" unit, b) a "last to execute" unit , and c) a "first to execute" unit. Numbers display the state counter of each unit.

- $t + \tau + 2T$ is the earliest possible transition execution time: it corresponds to a unit which would have run just after the first write; this unit just waits for two periods before executing,
- and finally, $t + \tau + 3T + \tau'$ is the time at which this unit would possibly write again on the bus; it is the earliest new write on the bus.

This clearly shows that:

The earliest transition execution strictly precedes the latest write, and the latest execution strictly precedes the earliest next write. Thus, phases of input acquisitions and transition executions are disjoint. Hence every unit executes with coherent inputs. It is easy to show that the same property applies to state values.

Figure 4.8 shows typical situations which illustrate the proof.

4.4.3 Interest of this Algorithm

In our opinion, this algorithm presents several interests.

First, it provides a smooth synchronization which operates on need: if nothing changes either in the environment or in the state variables of the units, no synchronization takes place. In particular, if stable systems are considered, this algorithm will not maintain a useless absolute time.

It is itself robust in that it only operates by periodically reading the bus input buffers, which is a non blocking operation, and does not require any hand-shake. The only required knowledge is made of the unit periods.

It is fairly general in that it matches any non stable, non robust system.

It requires a speed-up of order 4 which can be expected to be fairly efficient.

Chapter 5

From Synchronous Validation to Quasi-Synchronous Execution

In this chapter, we try to draw the practical consequences of the previous one. We first show that a control system validation (simulation, test, formal verification etc.), even in the centralized case, still requires a robustness analysis about the relations between the controller and its environment. Then we extend these requirements to the distributed case. A special section is devoted to the concurrent case, where programs are distributed as processes within the same computer.

5.1 From Synchronous Validation ...

By validation we understand any of the usual means by which engineers assess the correct functioning of their systems, either simulation, test or even formal verification.

The usual diagram on which validation is performed is the one of figure 4.4, which can be extended from the continuous case to every other validation case, even the logical one, thanks to the following correspondence table:

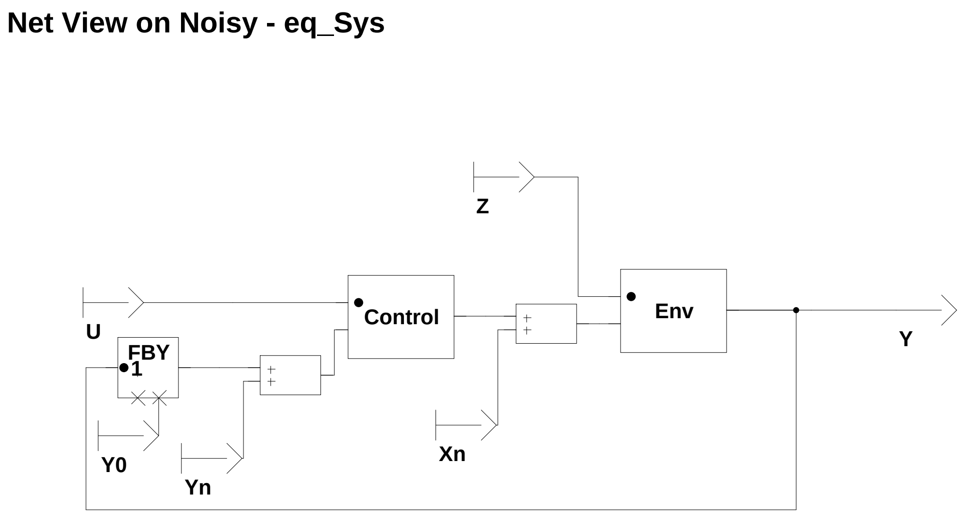


Figure 5.1: A Control system with sampling errors: Yn and Xn represent additive noises.

	continuous	logical
U	set values	operator inputs
X	control	control
Z	perturbations	oracles accounting for non determinism
Y	measurements	observations and property truth value

However, it seems to us that this synchronous validation diagram must be handled with care since it represents an abstraction from the "real world" that requires being justified.

5.1.1 Continuous Systems

When dealing with continuous systems, this abstraction neglects sampling and measurement errors. A better and more faithful validation scheme should take into account these errors. Figure 5.1 displays such a scheme.

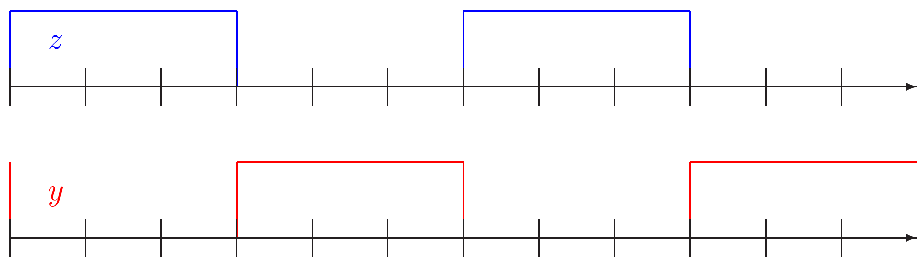


Figure 5.2: A non robust mutual exclusion

5.1.2 Non Continuous Systems

When dealing with logical and hybrid systems, an analog phenomenon takes place, in the sense that perfect synchrony validation neglects real-time problems. For instance, assume a system requires the mutual exclusion property that y and z never hold at the same time:

$$\texttt{always not } (y \texttt{ and } z)$$

Would one be satisfied with the system ?

$$\begin{array}{lcl} system(u) & = & y, z \\ \texttt{where } y & = & u \\ \texttt{and } z & = & \texttt{not } u \end{array}$$

Formally speaking, proving that the system satisfies the property is obvious, but clearly this system does not robustly fulfill it, because, if we look at things more deeply, it may not be the case that y and z will always be written at the same time and, if it is not the case, there will be some small time intervals where both y and z hold simultaneously. Figure 5.2 displays such a non robust mutual exclusion.

So what worth is this formal proof? And which is a faithful validation scheme, analog to the one of figure 5.1 for continuous systems?

Answering these questions is however more difficult than in the continuous system case, because there are several possible answers:

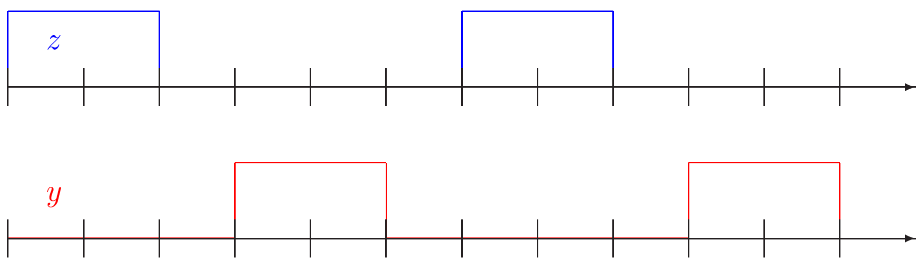


Figure 5.3: A robust mutual exclusion

- First we can admit that there exist small time intervals where the required property can be violated. That is to say we accept non robust systems and properties because we know for some (maybe physical) reason that property violation for small time intervals is not important. But the question in this context is: how can we take into account this fact when validating distributed systems?

- Else, we can decide that this property is not robust and modify it for instance by saying that between successive `true` values of y or z, there should exist some time interval where none of the variables is `true`. Then, the proposed system will not satisfy the property and we have to modify it. Figure 5.3 shows such a robust mutual exclusion.

- Else, we modify the way we model the relations between the system and its environment, so as to take into account more realistically real-time problems. Then, here also, the proposed system will not satisfy the property and will have to be modified in the sense of a better robustness.

Adopting this later solution consists in replacing, in the validation model for continuous systems, additive errors with non deterministic bounded delays. Figure 5.4 displays such a model.

In this figure, the `Bounded_delay` Scade assertion is intended to capture the fact that the computer does not answer in zero time, but sends outputs to the environment at some arbitrary time during its computing cycle. Similarly

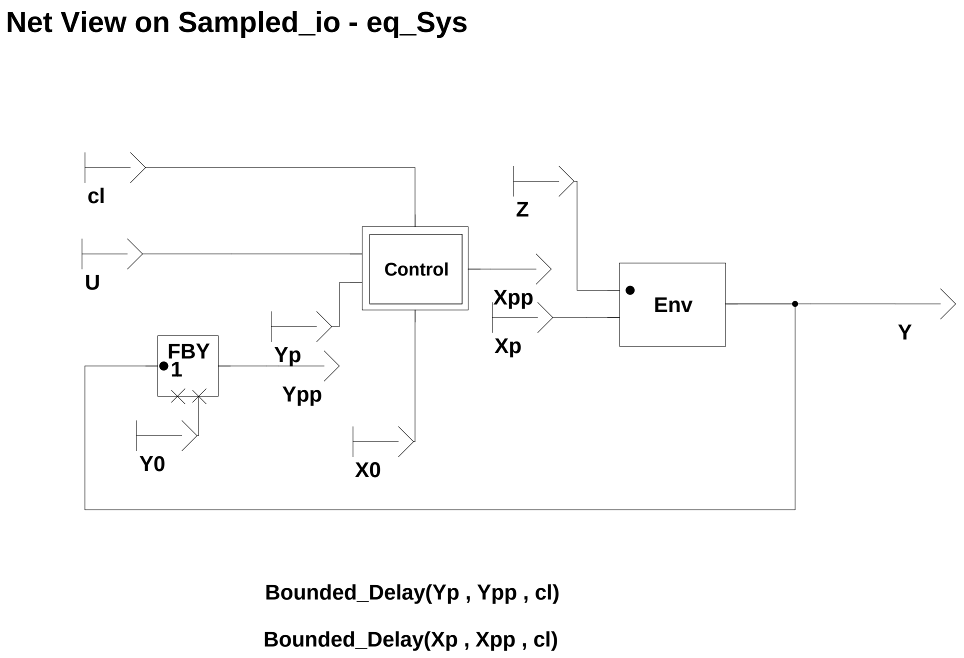


Figure 5.4: A control system with sampling delays: Yp and Xp represent pseudo-inputs and outputs (oracles).

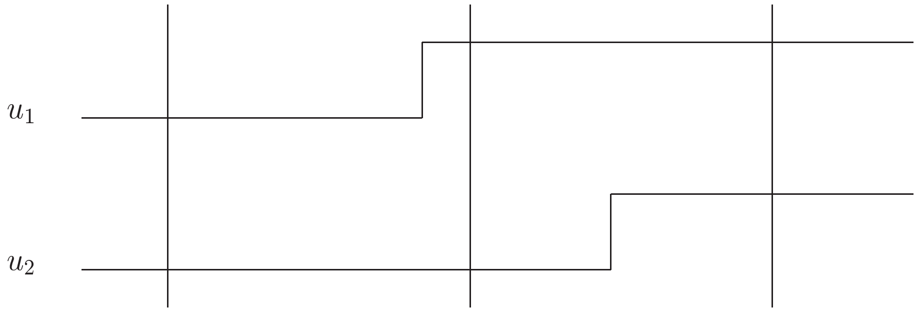


Figure 5.5: Bounded non deterministic delay

inputs are sampled at some arbitrary time during the preceding computing cycle.

Definition 13 (Bounded delay assertion).

$$\begin{array}{rcll} Bounded\_Delay(u_1, u_2, c) & = & & Sample_1(c, u_1) \\ & & and & Sample_1(c, u_2) \\ & & and & u_2 \texttt{ when } c = u_1 \texttt{ fby } (u_1 \texttt{ when } c) \end{array}$$

This definition is illustrated at figure 5.5. It uses the $Sample_1$ assertion which says that a boolean signal u has a minimum stable interval T_u larger than the maximum period of clock c.

Validating a property on this validation model ensures that the couple (system, environment) is robust with respect to that property. Clearly, our quasi-synchronous distributed architecture can only cope with this kind of property. Thus, we shall assume in the sequel that it is actually the case.

5.2 . . . To Quasi-Synchronous Execution

This section tries to draw the practical consequences of the preceding chapters. Three cases are considered: the continuous, the non continuous and the mixed ones.

5.2.1 The Continuous Case

This case is quite clear. A recipe can be:

- compute the equivalent errors on inputs arising from distribution;
- inject these errors in the centralized validation model;
- validate the system.

Then you can be confident that the properties you find on this model will be kept by quasi-synchronous execution.

5.2.2 The Non Continuous Case

The case, here, is more involved. It can be summarized as follows:

- validate
  - either the centralized model with non deterministic bounded delays on inputs and outputs,
  - or the centralized ideal model , if you have good reasons of assuming that the properties you check are insensitive to bounded delays;
- check the robustness of your given partition;
  - if it is robust, you can implement it as is, by taking a seep-up equal to the maximum number of location traversed by the dependency graph;[1]
  - else, you can use the synchronization algorithm and take a speed-up equal to 4.

5.2.3 The Mixed Continuous-Non Continuous Case

This case raises the problem of communication between the two parts.

The discrete to continuous part does not seem to raise problems: it consists in general of switches allowing the choice of a continuous output out of several continuous inputs. These choices have to be taken into account in the numerical analysis of computing errors.

[1]Because each traversal may loose a sample.

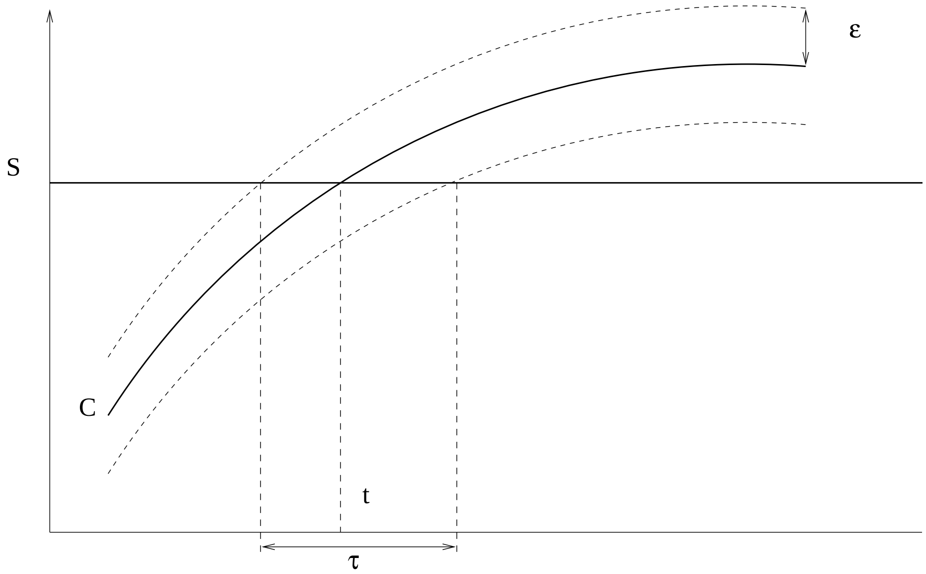


Figure 5.6: Comparing a signal to a threshold.

The continuous to discrete part is more involved: it is mainly based on comparison operators. The problem here is to take into account two contradictory facts:

- continuous computations are not perfectly accurate;
- discrete signals must have known bounded delays.

Figure 5.6 depicts a typical situation where a signal C is compared to a threshold S. It illustrates the well-known "threshold crossing" effect: the discrete delay is linked to the continuous error through the signal derivative at the crossing point.

$$\tau = \frac{2\epsilon}{|C'(t)|}$$

If this derivative vanishes, the corresponding delay may grow up to infinity.

In our opinion, this kind of analysis is part of the system engineering analysis and should not be avoided.[2]

[2] Computer Science should not take in charge what does not belong to its domain.

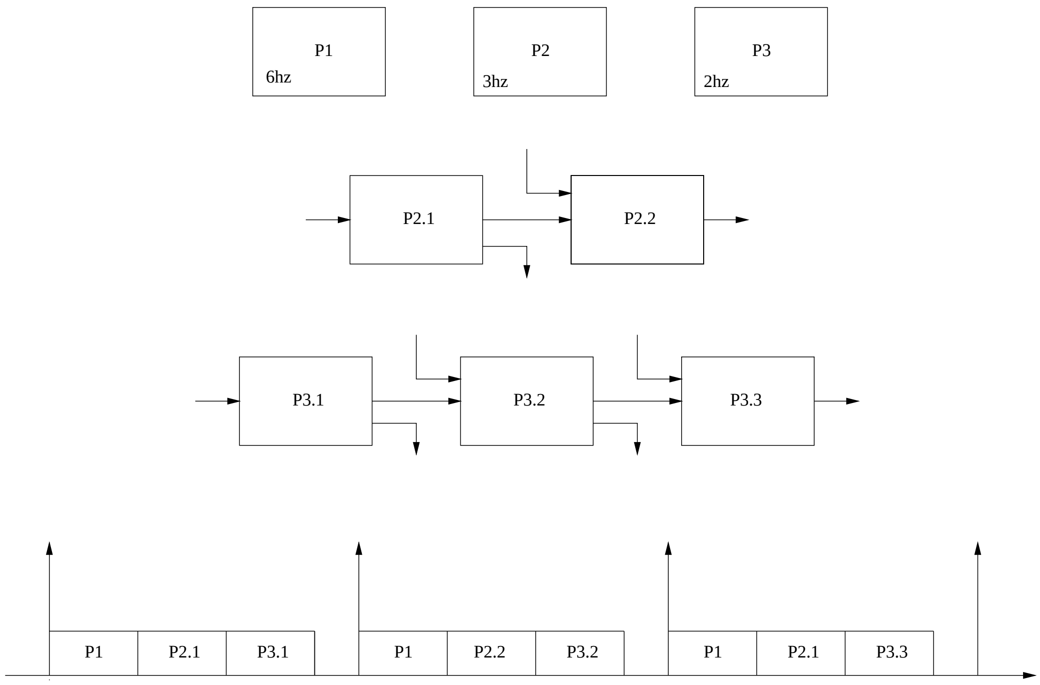


Figure 5.7: Actual Practices: processes P2 and P3 are split into parts (a and b) so as to get the the scheduling (c)

5.3 Concurrency

In this section, we investigate the application of techniques proposed in this report to the concurrent implementation of synchronous programs.

5.3.1 Actual Practices

In the field of critical systems, preemptive scheduling is seldom used: the usual practice, for instance at Aérospatiale is as follows (figure 5.7): when processes of different frequencies are to be implemented in a single computer, processes of lower frequency are split into parts and scheduled manually so as to yield a single synchronous program with conditions allowing to determine which sub-processes are to be executed at each cycle. The only condition required is that frequencies have rational ratios. Then, the real-time clock frequency of the single synchronous periodic program is the least common multiple of the process frequencies.

The interest of such an approach is to yield deterministic programs, to avoid using real-time executives, and to allow finely tuning response times. However, it has the drawback of needing a human interaction which may not be very effective and can be error prone. Finally, it does not optimally use computing resources.

5.3.2 A Crisys Proposal

The idea would be to deal with concurrent processes as if they were distributed ones. This entails first a need for a scheduling strategy for executing these processes on the same computer and then a method for letting them communicate.

Scheduling

The earliest dead-line first (EDF) preemptive scheduling policy seems a good choice: due to the fact that frequency ratios are rational, it is at least as predictable as periodic synchronous programming: a necessary and sufficient condition [18] is:

$$\sum_{i=1,n} \frac{C_i}{T_i} \leq 1$$

where n is the number of processes and C_i and T_i are the maximum execution time and period of process i. This condition is exactly the one that arises in the case of a single periodic process:

$$\frac{C}{T} \leq 1$$

Figure 5.8 illustrates this scheduling policy in the same case as in figure 5.7.

Communication

This scheduling actually supports two types of communication, an exact one, which preserves the functional semantic of SCADE and thus preserves properties, but maximizes latency, and an approximate one, which preserves properties, but minimizes latency:

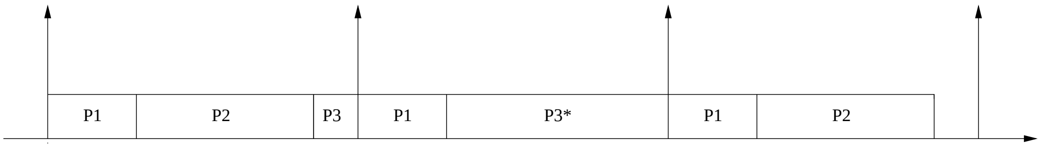


Figure 5.8: Dead-line driven scheduling in the case of figure 5.7: a “*” means the continuation of an interrupted execution.

Exact Functional Communication This exact functional approach arises from the very definition of SCADE primitives. Let us illustrate it by an example:

Let c be a half rate clock, x a variable existing at the full rate process, and f some function that we want to apply to x at half rate.

Two cases can appear:

- We need the result immediately. This amounts to computing the expression
$$\texttt{current}(z_0, c)\ f(x\ \texttt{when}\ c)$$
and there is no way to concurrent execution, because, at each cycle of the fast clock,the result is needed as shown in the following diagram:

c	t	f	t	f	t
x	x_0	x_1	x_2	x_3	x_4
x `when` c	x_0		x_2		x_4
$f(x$ `when` $c)$	$f(x_0)$		$f(x_2)$		$f(x_4)$
`current`(z_0, c) $f(x$ `when` $c)$	$f(x_0)$	$f(x_0)$	$f(x_2)$	$f(x_2)$	$f(x_4)$

- We either do not need the result at the fast clock rate or we need it but with at least a *unit delay of the slow clock*. Then concurrent executions are allowed, which keep the semantic of SCADE.

c	t	f	t	f	t
x	x_0	x_1	x_2	x_3	x_4
x `when` c	x_0		x_2		x_4
$z = f(x$ `when` $c)$	$f(x_0)$		$f(x_2)$		$f(x_4)$
z_0 `fby` z	z_0		$f(x_0)$		$f(x_2)$
`current`(z_0, c) z_0 `fby` z	z_0	z_0	$f(x_0)$	$f(x_0)$	$f(x_2)$

As we can see, we have two cycles of the fast clock for computing the f function before its result is needed.

This example can be generalized as follows: functional semantic is preserved if:

- Processes communicate by shared memory.
- Processes read their input values from the shared memory as soon as they are scheduled and write their outputs in the shared memory when their cycle is completed.

Latency minimization However, exact semantic has the drawback of maximizing delays because values are read as soon as possible and written as late as possible. An alternative solution which tends to minimize delays would be to read only when needed and to write as soon as possible (i.e. when data are available). Yet, this solution will in general look like the one of distribution and will not preserve synchronous properties but on some conditions, for instance order insensitivity and confluence[3].

Requirements for code generation

Thus requirements for concurrent code generation are quite simpler than in the distributed case:

- Group synchronous specifications according to their frequencies and compile sequential code for each of them.
- If exact semantic is chosen check that communication from low frequency tasks to high frequency ones passes through a low frequency unit delay.
- If latency minimization is chosen, check for robustness.

[3]Stability is not mentioned here because in the concurrent setting absolute time is preserved.

Chapter 6

Fault Tolerance

We consider here fault tolerance. For continuous control, this leads us to threshold voting: two signals are considered to disagree if they differ for more than the maximum normal error. Similarly, combinational computations yield bounded-delay voting: two signals are considered to disagree if they remain different for more than the maximum normal delay. Extending this scheme to apply to sequential functions by transforming them into combinational ones, that is to say by bounded-delay voting on the state, leads to problems of Byzantine faults. However, we show here that 2/2 voting schemes are not sensitive to Byzantine faults and behave properly. This provides us with self-checking schemes which can then be used to build fault tolerant strategies by means of selective redundancy.

6.1 Threshold Voting for Continuous Functions

Knowing bounds on the normal deviation between values that should be equal, easily allows the design of threshold voters. For instance, a 2/3 voter for scalar values can be written:

$$
\begin{array}{rll}
voter2/3(x_1, x_2, x_3, \epsilon) = & \texttt{if} & |x_2 - x_1| \leq \epsilon \\
& \quad or & |x_3 - x_2| \leq \epsilon \\
& \quad or & |x_1 - x_3| \leq \epsilon \\
& \multicolumn{2}{l}{\texttt{then}\ median(x_1, x_2, x_3)} \\
& \multicolumn{2}{l}{\texttt{else}\ alarm}
\end{array}
$$

where

$$median(x_1, x_2, x_3) = max(min(x_1, x_2), min(x_2, x_3), min(x_3, x_1))$$

6.2 Bounded Delay Voting for Combinational Functions

Let us consider several copies of a boolean signal x, received by some unit of period T with a maximum normal delay τ_x such that $\tau_x + T_M < T_x$. Then,

- the maximum time interval where two correct copies may continuously disagree is obviously τ_x,
- the maximum number of samples where two correct copies continuously disagree is
$$nmax = E(\frac{\tau_x}{T_m}) + 1$$

This allows us to design **bounded-delay voters** for bounded-delay booleans signals. For instance, a 2/2 voter could be:

Definition 14 (2/2 bounded-delay voter).

$$\begin{array}{lcl} voter2/2(x_1, x_2, nmax) & = & x \\ \\ \texttt{where}\ x,\ \ n & = & \texttt{if}\ x_2 = x_1 \\ & & \texttt{then}\ x_1,\ \ 0 \\ & & \texttt{else}\ \ \texttt{if}\ 0\ \texttt{fby}\ n < nmax - 1 \\ & & \qquad \texttt{then}\ x0\ \texttt{fby}\ x,\ \ 0\ \texttt{fby}\ n + 1 \\ & & \qquad \texttt{else}\ alarm \end{array}$$

- this voter maintains a counter n with initial value 0, and its previous output, with some known initial value $x0$,
- whenever the two inputs agree , it outputs one input and resets the counter,
- else, if the counter has not reached $nmax - 1$, it increments it and outputs the previous output,

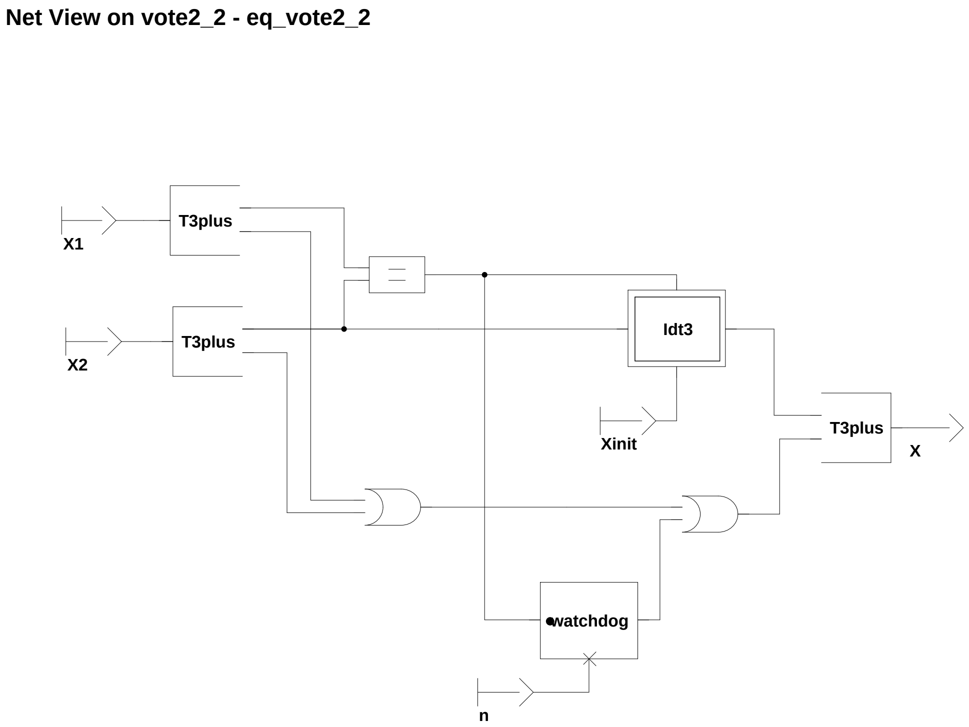


Figure 6.1: A 2/2 bounded delay voter for combinational systems

- else it raises an alarm.

Figure 6.1 displays the Scade implementation of this voter.

Theorem 6.2.1. *$voter2/2$ raises an alarm if the two inputs disagree for more than $nmaxT_M$ and otherwise delivers the correct value with maximum delay $(nmax + 1)T_M$.*

6.2.1 Bounded Delay Voting on Tuples

Combining bounded delay voting on booleans and confirmation functions allows for the definition of voters for tuples and combinational functions:

Definition 15 (2/2 bounded-delay voter for tuples).

$$\begin{aligned} Voter2/2(X_1, X_2, nmax) &= confirm(X'', nmax) \\ \texttt{where}\ \forall i \in \{1, n\} : x''_i &= voter2/2(x_{1,i}, x_{2,i}, nmax) \end{aligned}$$

Assuming:

- $X_i, i = 1, 2$ are τ_x bounded delay images of the same X,
- each component of X has minimum stable time interval T_x
- $nmax = E(\frac{\tau_x}{T_m}) + 1$
- $T_x > (n.nmax + 1)T_M$

yields:

Theorem 6.2.2. *$Voter2/2$ raises an alarm if any corresponding components of the two inputs disagree for more than $nmaxT_M$ and otherwise delivers the correct value with maximum delay $(n.nmax + 1)T_M$.*

This is the combination of theorems 4.2.3 and 6.2.1, but for the fact that we do not need to propagate the +1 additional delay from the voter to the confirmation function, because both occur in the same computing unit.

6.3 Sequential Functions

6.3.1 Bounded-Delay Voting for Sequential Functions

An idea for applying bounded-delay voting to sequential functions would be to transform sequential functions into combinational ones by setting apart state memorization and voting on states as well as on inputs:

Instead of computing:

$$x_i = F(x0 \texttt{ fby } x_i, vote(U))$$

compute

$$x_i = F(vote(X0 \texttt{ fby } X), vote(U))$$

This does not work in general, for Byzantine-like reasons: our bounded-delay voters are sequential systems in the sense that they store as state variable the last value on which units agreed; then a malignant unit may successively agree with states of correct units, while these units disagree for delay reasons, leading the state voters of these correct units to store incoherent values.

6.3.2 2/2 Vote for Sequential Functions

Quite surprisingly, this phenomenon does not seem to appear for 2/2 bounded delay vote. 2/2 voting is in general not sensitive to Byzantine behaviors because a malignant unit only communicates with a correct one: either it agrees with it and behaves like the correct one or it disagrees and the fault is detected. This property of avoiding Byzantine problems may explain why this fault-detection strategy is very popular.

Consider the **voting scheme**:

$$\begin{aligned}
X_1 = F(\ & vote2/2(X0 \texttt{ fby } X_1, X_2', n(nmax_u + nmax_X)),\\
& confirm(\ \{vote2/2(u_{1,i}', u_{2,i}', nmax_u), i = 1, n\},\\
& \qquad\qquad nmax_u + nmax_X))\\
X_2 = F(\ & vote2/2(X0 \texttt{ fby } X_2, X_1', n(nmax_u + nmax_X)),\\
& confirm(\ \{vote2/2(u_{1,i}'', u_{2,i}'', nmax_u), i = 1, n\},\\
& \qquad\qquad nmax_u + nmax_X))
\end{aligned}$$

where

- F is a 1_stable transition function.
- U'_i and U''_i are n_tuples linked to some U by bounded delays less than τ_u and $nmax_u = E(\frac{\tau_u}{T_m}) + 1$.
- X'_i are linked to X_i by bounded delays less than τ_X, and $nmax_X = E(\frac{\tau_X}{T_m}) + 1$.
- we assume $\tau_X \leq \tau_u$
- the minimum stable interval of each component of U, T_u is larger than $(n.(nmax_u + nmax_X) + 1)T_M$.

Remark: we use here our simple boolean votes extended to tuples and not our tuple voters. The reasons are that, for states, we do not need confirmation, because one tuple comes from the very unit which computes the vote and the other one comes in one shot from the other unit, and for inputs, we set apart the confirmation because we need a longer stable delay.

Theorem 6.3.1. *In the absence of faults, any state voter delivers a correct state with maximum delay* $(n.(nmax_u+nmax_X)+1)T_M$ *and delivers an alarm if some input components disagree for more than* $nmax_uT_M$ *or one of the two units does not compute as specified for more than* $n.(nmax_u + nmax_X)T_M$.

Proof: The proof is by induction on an execution path: Initially, inputs and states agree. Assume a time instant where inputs and states agree. From the stability assumption, nothing will change unless inputs change. Assume some input changes. It takes at most $(nmax_u+1)T$ for both units to agree on this input change and at most $(nmax_x + 1)T$ to agree on the corresponding state change. If this is the case we are done. Now it may be the case that meanwhile another input component changes. It the agreement on states has not been reached, no harm is done because, in both units, state voters still maintain the old state. If the agreement on states is reached in one unit, it will be certainly reached in the other one, because the "confirm" function freezes inputs for a sufficient delay for reaching an agreement in both units (this part of the proof is illustrated at figure 6.2). By induction on the number of components, if a state agreement has not been reached before, it will be reached when every input component have changed once, because no

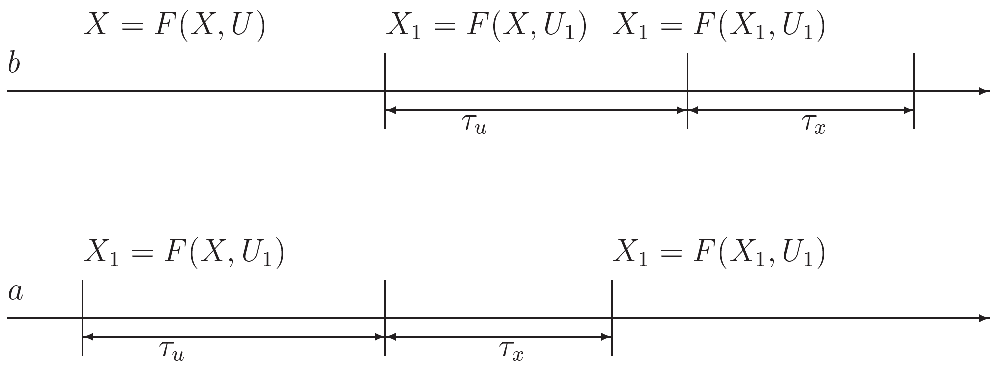


Figure 6.2: Proof illustration: process a sees the input changing from U to U_1 first and computes X_1. Then process b sees the same change and also computes X_1. Having received X_1 from a, its state voter moves to X_1 but, for the sake of stability, it goes on computing X_1. Now this value will in turn reach a before the input changes again, thus allowing state voters to reach an agreement on X_1.

component can change again from the minimum stable interval assumption. Finally the alarm part of the theorem is obvious from the voting algorithm.

Figure 6.3 displays a Scade implementation of such a voter.

6.3.3 From Fault Detection to Fault Tolerance

2/2 voters account for fault detection and for the design of self-checking modules. Now these modules can be combined by means of selective redundancy so as to form fault-tolerant modules. In the Airbus architecture, this goal is achieved thanks to a global system approach: two self-checking computing systems are provided, each one being able to control the aircraft by itself.

Another possibility is to design hybrid redundancy voters: a primary self-checking system operates until it fails. Meanwhile a secondary system votes on the states of the primary one, so as to stay coherent with it. When the primary system fails, the control is moved to the secondary one which is hopefully still correct.

When designing it, one has to solve the following problem: we switch from the primary system to the secondary one when the watchdog counting the number of disagreements has expired. But then, it may be the case that,

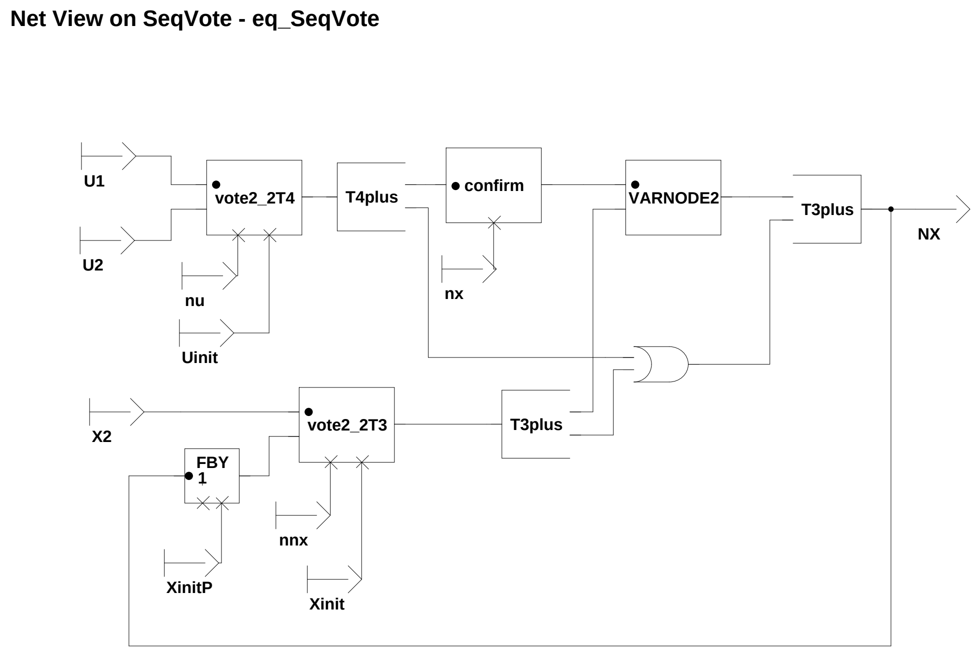


Figure 6.3: A 2/2 bounded delay voter for sequential systems

because of asynchrony, the secondary system is not yet coherent. We must not raise an alarm here, but, on the contrary leave this system the time to reach a coherent state. This is achieved by the following resetting 2/2 voter:

Definition 16 (2/2 Resetting Voter).

$$
\begin{array}{lcl}
rvoter2/2(x_1, x_2, reset, nmax) & = & x \\
\texttt{where}\ x & = & \texttt{if}\ x_2 = x_1 \\
 & & \texttt{then}\ x_1 \\
 & & \texttt{else}\ \ \texttt{if}\ 0\ \texttt{fby}\ n < nmax - 1 \\
 & & \qquad\ \ \texttt{then}\ x0\ \texttt{fby}\ x \\
 & & \qquad\ \ \texttt{else}\ alarm \\
\texttt{and}\ n & = & \texttt{if}\ x_2 = x_1 \vee reset \\
 & & \texttt{then}\ 0 \\
 & & \texttt{else}\ \ \texttt{if}\ 0\ \texttt{fby}\ n < nmax - 1 \\
 & & \qquad\ \ \texttt{then}\ 0\ \texttt{fby}\ n + 1 \\
 & & \qquad\ \ \texttt{else}\ alarm
\end{array}
$$

We are now in a position to design the hybrid voter:

Definition 17 ($1/2 \times 2/2$Voter).

$$
\begin{array}{lcl}
voter1/2 \times 2/2(X_1, X_2, X_3, X_4, nmax) & = & X \\
\texttt{where}\ X & = & rvoter2/2(X'_1, X'_2, reset, nmax) \\
\texttt{and}\ X'_1,\ \ X'_2 & = & \ \texttt{if}\ primary \\
 & & \ \texttt{then}\ X_1,\ \ X_2 \\
 & & \ \texttt{else}\ X_3,\ \ X_4 \\
\texttt{and}\ primary & = & true\ \texttt{fby}\ \ \texttt{if}\ X = alarm \\
 & & \qquad\qquad\ \ \texttt{then}\ false \\
 & & \qquad\qquad\ \ \texttt{else}\ primary \\
\texttt{and}\ reset & = & (\neg primary) \wedge (true\ \texttt{fby}\ primary)
\end{array}
$$

- *primary* is initially true and the vote is performed on units 1 and 2.
- When one of these fails, primary becomes false for ever and the vote is performed on units 3 and 4.

Chapter 7

Conclusion

We thus have shown how to implement continuous, combinational and some robust sequential functions on the quasi-synchronous architecture. Finally, we have been able to provide interesting fault tolerant schemes only based on the timing properties of periodic unsynchronized systems. This was quite easy to do for continuous and combinational systems. The problem was more involved for sequential stable systems, but, nevertheless we have found a fault detection scheme which applies to this case and is still only based on timing properties. This allows self-checking dual modules to be build that can serve as building blocks for more elaborated fault tolerant strategies.

One quite obvious possible use of this work is to help certification authorities in getting a better insight on these techniques which are still in use on many systems. Another possible one would be to help designers in choosing between clock synchronization techniques and the ones presented above: one outcome of this work is the computation of periods depending on some characteristics of the application, mainly the error and minimum stable time interval of the inputs, and the data-flow and robustness properties of the distribution. This has clearly to be balanced with the cost of clock synchronization.

A question left open here is the one of building robust systems within the synchronous framework. Solving this problem would allow us to avoid either checking for robustness, which is clearly a reverse engineering technique, or relying in the designer's skill.

Another question left open in this work is the one of unstable sequential systems, for which our techniques for fault tolerance clearly do not apply. But, this raises the question of when the implementation of critical control systems do require programming unstable systems. Our opinion is that, as for continuous control, this is seldom the case. More generally, the celebrated "Y2K" bug tells us that unstable systems should be avoided as most as possible.

Last but not least, we hope to attract the attention of computer scientist on this kind of techniques which seem to have been somewhat neglected in the past. It is our opinion that there is by now some revival of the asynchronous circuit culture to which this work may participate.

Bibliography

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[2] R. Alur and D.L.Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] S. Bensalem, P. Caspi, C. Dumas, and C. Parent-Vigouroux. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7, San Jose*. IEEE Computer Society, January 1999.

[4] J.-L. Bergerand and E. Pilaud. Saga : A software development environment for dependability in automatic control. In *IFAC-SAFECOMP'88*. Pergamon Press, 1988.

[5] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating reactive processes. In *20th Principles of Programming Languages ACM Conference*, pages 85–89, January 1993.

[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.

[7] J. A. Brzozowski. Delay-insensitivity and ternary simulation. *Theoretical Computer Science*, 1998. to appear.

[8] J. A. Brzozowski and C-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[9] P. Caspi and N. Halbwachs. A functional model for describing and reasoning about time behaviour of computing systems. *Acta Informatica*, 22:595–627, 1986.

[10] A. Chatha. Fieldbus: The foundation for field control systems. *Control Engineering*, pages 47–50, May 1994.

[11] M.J. Fisher, N.A. Lynch, and M.S. Patterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[12] 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.

[13] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[14] Cl. Hennebert and D. Lancien. SACEM : les choix fondamentaux. *Revue Générale des Chemins de Fer*, 109(6):19–21, 1990.

[15] C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–676, 1978.

[16] A.H. Hopkins, T. Basil Smith, and J.H. Lala. FTMP:a highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239, 1978.

[17] 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.

[18] C.L. Liu and J.W. Layland. Scheduling algoritms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[19] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[20] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, July 1983.

[21] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92.* North-Holland, 1994.

[22] M. Pease, R.E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–237, 1980.

[23] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)* Paderborn, Germany. LNCS 1099, Springer Verlag, July 1996.

[24] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Lewitt, P.M. Melliar-Smith, R.E Shostak, and Ch.B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, 1978.