Threshold and Bounded-Delay Voting in Critical Control Systems*

Paul Caspi and Rym Salem
name@imag.fr

Laboratoire Verimag (CNRS, UJF, INPG)
http://www-verimag.imag.fr

Abstract. This paper investigates the possibility of implementing fault tolerance in control systems without using clock synchronization. We first show that threshold voting applies to stable continuous systems and that bounded delay voting applies to combinational systems. We also show that 2/2 bounded delay voting is insensitive to Byzantine faults and applies to stable sequential systems. It thus allows the implementation of hybrid fault tolerance strategies.

1 Introduction

It seems that, from the very early times of SIFT and FTMP [10, 7], 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 [9], and produced an important academic activity [6], culminating in the time-triggered approach to the development of these systems [8].

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 [4, 3]. 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 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 [4], nuclear plants [3], etc.)

This paper aims at providing some answers to this question. It will be organized as follows: first we shall look at the architectural evolution from analog boards to distributed control systems. Then, we shall consider continuous control. This will lead us to the concept of threshold voting, based on accuracy

* This work has been partially supported by the CRISYS Esprit Project

estimates: two signals are considered to disagree if they differ for more than the maximum normal error. Then we shall 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 and yields 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.

2 The Architectural Evolution

Analog/digital communication: Aircraft control systems illustrate this evolution which can also be found in many other fields of industrial control: 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.

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.

Field busses: then for the sake of optimization, these serial links are replaced by busses of several standards, (aircraft, industrial, 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.

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 [9], 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.

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

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 consecutive failed receives whose probability is not negligible with respect to fault tolerant requirements (in the case of error correction, $n = 1$). 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 $|t - t'| \leq T_{rM} + nT_{sM}$*

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*

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

3 Continuous Control

3.1 Continuous Signals and Functions

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

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

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

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

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

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

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.

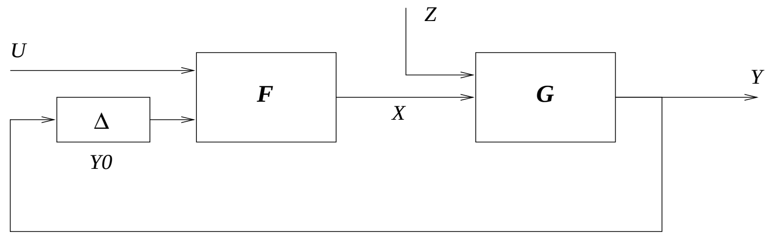

Fig. 1. A Control System

The stabilized case Unfortunately, controllers are often not stable: a typical situation is shown at figure 1. This closed-loop system computes the following functional equations: $X = F(U, \Delta_{Y0}^T Y)$ and $Y = G(X, Z)$, where F is the controller, G 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. Δ is the unit delay for periodically sampled signals[1]:

$$\begin{aligned}\Delta_{x0}^T x(nT) = \ & \texttt{if } n = 0 \\ & \texttt{then } x0 \\ & \texttt{else } x((n-1)T)\end{aligned}$$

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. 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. 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 [2].

[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$_X$.

3.3 Threshold Voting

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:

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

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

Notations: In the sequel, algorithms are expressed by abstracting over time indices, thus, $x_1 = x_2$ means $\forall n \in N : x_1(nT) = x_2(nT)$.

4 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 will help in recovering it.

4.1 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 version of it, a possible definition, which only presents a small deviation from the one in [5], being as follows:

Definition 3. *A signal $x \in R^+ \to B^n$ has* **uniform bounded variability** *if it is right-continuous and 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 $x(t^-)$ represents the left limit 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* $c_x(T_x) = 1$*:* $T_x = sup\{T \mid c_x(T) \leq 1\}$

Thus the sampling period should be smaller than T_x. We can now relate bounded delays and minimum stable time interval:

Theorem 2 (Bounded delay and minimum stable time interval). *If* x' *is a* τ *bounded delay image of* x*, if* T_x *is the minimum stable time interval of* x *and if* $\tau < T_x$ *then the minimum stable time interval of* x' *is:* $T_{x'} = T_x - \tau$.

Remark: this result can be improved by considering both minimum τ_m and maximum τ_M delays: $T_{x'} = T_x - (\tau_M - \tau_m)$ 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 3. *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.*

4.2 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 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.

Fortunately, some filtering procedures make it possible to change incoherent delays into coherent ones: these are 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: $\forall i = 1, n, \exists t'_i, \forall t, 0 \leq t - t'_i(t) \leq \tau_X : x'_i(t) = x_i(t'_i(t))$ where we assume that all delays have the same upper bound τ_X. We consider the following confirmation function:

Definition 6 (Confirmation function).

$$
\begin{array}{lll}
\mathit{confirm}(X', nmax) & = & X'' \\
\texttt{where } X'',\ \ n & = & \texttt{if } X' \neq \Delta^T_{X0} X' \\
 & & \texttt{then } \Delta^T_{X0} X'',\ \ 0 \\
 & & \texttt{else if } \Delta^T_0 n < nmax - 1 \\
 & & \quad\quad \texttt{then } \Delta^T_{X0} X'',\ \ \Delta^T_0 n + 1 \\
 & & \quad\quad \texttt{else } X',\ \ \Delta^T_0 n
\end{array}
$$

– 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. *The confirm function outputs a delayed image of the original tuple:* $\exists t', \forall t, 0 \leq t - t'(t) \leq (n.nmax + 1)T : \mathit{confirm}(X', nmax)(t) = X(t'(t))$

This shows that incoherences due to variable delays can be changed into coherent delays. Once this coherent delay problem is solved, we can consider bounded delay voting. 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 Bounded Delay Voting

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 7 (2/2 bounded-delay voter).

$$
\begin{array}{lll}
voter2/2(x_1, x_2, nmax) = x & & \\
\texttt{where } x,\ n & = & \texttt{if } x_2 = x_1 \\
& & \quad \texttt{then } x_1,\ 0 \\
& & \quad \texttt{else if } \Delta_0^T n < nmax - 1 \\
& & \quad\quad \texttt{then } \Delta_{x0}^T x,\ \ \Delta_0^T n + 1 \\
& & \quad\quad \texttt{else } alarm
\end{array}
$$

– 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,
– else it raises an alarm.

Theorem 5. *$voter2/2$ raises an alarm if inputs disagree for more than $nmaxT_M$ otherwise it delivers the correct value with maximum delay $(nmax + 1)T_M$.*

4.4 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 8 (2/2 bounded-delay voter for tuples).

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

Then the following theorem yields 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
– $nmax = E(\frac{\tau_x}{T_m}) + 1$
– $T_x > (n.nmax + 1)T_M$

Theorem 6. *$Voter2/2$ 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 and 5, 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.

5 Sequential Functions

For the sake of simplicity, we do not distinguish here between states and outputs. Hence, a sequential function $\mathcal{F}$ will be defined by its transition function F: $\mathcal{F}(U) = X$, where $X = F(\Delta^T_{X0} X, U)$

Definition 9. *A* **1_stable sequential function** *is a sequential function whose state only changes when its input changes. For all state X and input U:*
$F(F(X, U), U) = F(X, U)$

5.1 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, *i.e.* instead of computing $x_i = F(\Delta_{x0}^T x_i, vote(U))$ compute $x_i = F(vote(\Delta_{X0}^T X), vote(U))$.

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.

5.2 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. Let us consider the **voting scheme**:

$$\begin{aligned}
X_1 = F(\, & vote2/2(\Delta_{X0}^T X_1, X_2', n(nmax_u + nmax_X)),\\
& confirm(\{vote2/2(u_{1,i}', u_{2,i}', nmax_u), i = 1, n\},\\
& \quad nmax_u + nmax_X))\\
X_2 = F(\, & vote2/2(\Delta_{X0}^T X_2, X_1', n(nmax_u + nmax_X)),\\
& confirm(\{vote2/2(u_{1,i}'', u_{2,i}'', nmax_u), i = 1, n\},\\
& \quad nmax_u + nmax_X))
\end{aligned}$$

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

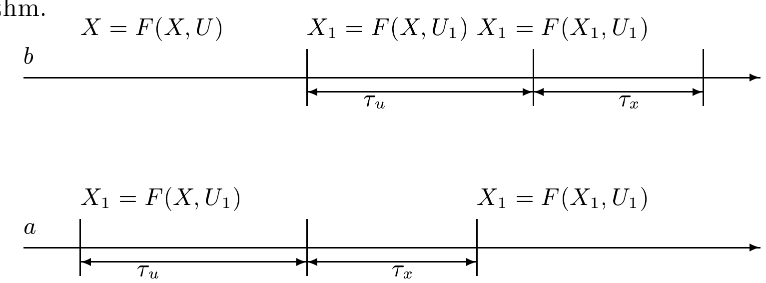

Fig. 2. Proof illustration: process a sees the input changing from U to U_1 first and computes X_1. Then process b sees the same change and also computes X_1. Having received X_1 from a, its state voter moves to X_1 but, for the sake of stability, it goes on computing X_1. Now this value will in turn reach a before the input changes again, thus allowing state voters to reach an agreement on X_1.

5.3 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, 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 10 (2/2 **Resetting Voter).**

$$
\begin{array}{ll}
rvoter2/2(x_1, x_2, reset, nmax) = x & \\
\texttt{where } x & = \texttt{if } x_2 = x_1 \\
& \quad \texttt{then } x_1 \\
& \quad \texttt{else if } \Delta_0^T n < nmax - 1 \\
& \qquad \texttt{then } \Delta_{x0}^T x \\
& \qquad \texttt{else } alarm \\
\texttt{and } n & = \texttt{if } x_2 = x_1 \vee reset \\
& \quad \texttt{then } 0 \\
& \quad \texttt{else if } \Delta_0^T n < nmax - 1 \\
& \qquad \texttt{then } \Delta_0^T n + 1 \\
& \qquad \texttt{else } alarm
\end{array}
$$

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

Definition 11 ($1/2 \times 2/2$**Voter).**

$$
\begin{array}{ll}
voter1/2 \times 2/2(X_1, X_2, X_3, X_4, nmax) = X & \\
\texttt{where } X & = rvoter2/2(X_1', X_2', reset, nmax) \\
\texttt{and } X_1',\ \ X_2' & = \texttt{if } primary \\
& \quad \texttt{then } X_1,\ \ X_2 \\
& \quad \texttt{else } X_3,\ \ X_4 \\
\texttt{and } primary & = \Delta_{true}^T \texttt{ if } X = alarm \\
& \qquad \texttt{then } false \\
& \qquad \texttt{else } primary \\
\texttt{and } reset & = (\neg primary) \wedge (\Delta_{true}^T primary)
\end{array}
$$

- *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.

6 Conclusion

Finally, it seems that 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 minimum stable time interval of the inputs, and the dimension of the involved tuples. This has clearly to be balanced with the cost of clock synchronization.

In this setting, a question left open in this work is the one of unstable sequential systems, for which our techniques 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.

References

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. 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.
3. J.-L. Bergerand and E. Pilaud. Saga : A software development environment for dependability in automatic control. In *IFAC-SAFECOMP'88*. Pergamon Press, 1988.
4. 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. P. Caspi and N. Halbwachs. A functional model for describing and reasoning about time behaviour of computing systems. *Acta Informatica*, 22:595–627, 1986.
6. 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.
7. A.H. Hopkins, T. Basi Smith, and J.H. Lala. FTMP:a highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239, 1978.
8. 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.
9. M. Pease, R.E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–237, 1980.
10. 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.