Some synchronization issues when designing embedded systems from components

Albert Benveniste[1]

Inria / Irisa, Campus de Beaulieu, 35042 Rennes, France ; Albert.Benveniste@inria.fr.
http://www.irisa.fr/sigma2/benveniste/home.html

Abstract This paper is sort of a confession. Issues of synchrony, asynchrony, and synchronization, arise frequently in designing embedded systems from components, like everyone I knew this for quite a long time. But it is only recently that it went aware of the *diversity* of such issues, depending on the context. The aim of this paper is to show and explain this diversity by looking at three areas where systems design is of interest, namely : 1/ building software or hardware architectures composed of components interacting asynchronously, 2/ synchronous hardware design from IP's, and 3/ designing distributed real-time control systems. A large part of this paper relies on other people's work, I indicate appropriate references in each case.

1 Think synchronously, act asynchronously

This is a widely used engineering practice in several important areas where embedded systems are designed.

Real-time control systems such as encountered in automobiles or even more in aeronautics, interact continuously with a physical system for the purpose of achieving some desired behaviour. The underlying physical system for control has its own dynamics. This dynamics is typically approximately modeled by some partial differential equation, or, more approximately, by some system of differential equations. As examples, think of car or aircraft engine combustion, car of aircraft dynamics, and so on. The important fact is that this dynamics is global and bound to physical time and space.

Analog control design consists in building another dynamical system to be put in closed loop with the physical one, in order to achieve a desired response to both external stimuli from the operator (the requests to be satisfied), and disturbances from the environment (against which robustness is required). Computer control, which has replaced the traditional analog control, aims at providing the same service, with a drastic improvement in flexibility and intelligence, but at the price of digitalization.

In doing his job, the designer handles global state and time, he thinks globally. Said in computer science terms, *the designer thinks "synchronously"*. The important fact to know about control engineering practice is that *the designer is aware that the model he uses is only approximate.* And this approximation occurs in both *time* and *space* :

- in *time*: sensing devices are only approximately synchronized, applying the computed control takes time, in other words, the (distributed) computer system acts with some (limited) asynchrony;
- in *space*: measurements are noisy, disturbances from the environment can be poorly known, models used are approximate (sometimes very coarsely so).

Until quite recently, the second problem (approximation in space) has overcomed the first one. The reason is that continuous control and regulation was the dominating task, for which approximation in time was solved using well established concepts from control such as that of *phase margin*[1]. However, real-time control systems have become much more complex by involving mode changes, protection, and more generally supervision and human-machine interaction. This results in a complex combination of continuous and discrete control, popularized recently as *hybrid* systems.

It turns out that the classical approach of control engineering to deal with approximation in time no longer suffices for the more complex embedded control systems. The lesson, first noticed and analyzed by Paul Caspi in [6], is that

> the synchrony/asynchrony issue, in distributed real-time embedded control systems, is motivated by the requested robustness against approximation in time.

In modern high performance hardware such as prevalent in embedded systems today, the combination of request for higher performance (small size, low power, high throughput and low latency), and low cost, calls for a design of systems from available components without paying (too much) for the corresponding overhead. Here we quote some arguments from L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli [5].

In hardware, while the number of gates reachable in a cycle will not change significantly and the on-chip bandwidth that wires provide will continue to grow, the percentage of the die reachable within one clock cycle will decrease dramatically: we will soon reach a point where more gates can be fit on a chip than can communicate in one clock cycle. In particular it has been predicted that for deep sub-micron designs a signal will need more than ten clock cycles to traverse the entire chip area. Also, it has been estimated that only a fraction of the chip area between 0.4% and 1.4% will be reachable in one clock cycle. This calls for a fine and efficient tuning of placement, routing, pipelining, and retiming. However, since precise data on wire-lengths are available only late in the design process, several costly re-designs become necessary to change the placement or the speed of the chip modules while satisfying performances and functionality constraints. Even worse, predefined Intellectual Properties (IP) cannot take into account such data, as these are inherently bound to the global system design.

[1] When designing a PID feedback loop, the designer provides robust control by ensuring some *gain margin* (to be robust against uncertainty in the gain and damping of the systems poles), and some *phase margin* (to be robust against uncertainty in the phase of the systems poles, or, equivalently, to be robust against various delays that can result from both the sensoring and computer equipments).

Still, low cost design of hardware calls for a design methodology that maintains the inherent simplicity of synchronous design, and yet does not suffer from the "interconnect-delay problem". The idea is that, at early stage of the design, both IP's and the system can be regarded as completely synchronous, i.e., just as a set of modules that communicate by means of channels having "zero-delay", i.e., a delay neglectible with respect to the period of the common "virtual" clock signal. And it is wished that, at later stages of the design where actual real clocks are used, with precise frequency and phase, proper retiming, pipelining, and resynchronization be automatically taken care of. In this way, successive re-designs are performed at the early design stage, where the simple synchronous model is assumed, and costly manual re-design phases are avoided. Note that the resulting hardware is still synchronous, we do not consider here the more advanced "asynchronous" technologies, in which the global synchronization of the different clocks is not maintained throughout the die. To summarize,

> the synchrony/asynchrony issue, in synchronous hardware systems design from IP's, consists in providing assistance for the design of proper retiming, pipelining, and resynchronization, which result from the "interconnect-delay problem".

Design of software systems from components is the prevalent approach, e.g., in telecommunications systems today. The migration, from a network dominant architecture to a service dominant one, results in an increase in Quality of Service (QoS) requirements. QoS is more and more taken care of by processing at the edges of the network, not within its backbone. The increase in versatility of services calls for far more dynamical protocol suites and frequent updates or upgrades of protocols. Also, software radio is a central technology for the 4th generation telecommunications. Thanks to ultra wide band, high performance, A/D converters, used in the front end of receivers, entire communication protocols can migrate to a fully digital technology. This makes it possible to dynamically and adaptively change the protocols at a receiver station. In the whole, this trend calls for drastic improvements in network management middleware and corresponding software architectures, in order to facilitate the reuse of predefined components. The essence of this point of view is that the systems software is made of Java/C/C++ *distributed* components interacting asynchronously via threading mechanisms, OS primitives, or software bus architectures.

The explosion of resulting middleware typically results in a huge overhead at run time. One of the techniques envisioned to face this is the combination of reflexive architectures [10][11] and specialization of pre-defined generic components (both off-line and on-line) [12][13].

However alternative approaches can be considered. While the above fully asynchronous point of view is perfectly valid when the major issue is that of resource sharing by different components which otherwise do not interact, it is no longer efficient at handling software or SW/HW systems, implementing, e.g., software radio techniques. In the latter case, the different components (filter banks, channel/source codes, protocols, schedulers) must effectively cooperate at delivering the overall expected function. Once more, the synchronous point of view can drastically simplify the early stages of the design, by providing easier specification, debugging and verification, and code optimization. Also, this point of view offers a smooth transition from a signal processing oriented style of specification, down to the system deployment.

The typical architecture is parallel and distributed (in the small), and involves a combination of DSP's, ASIC's, micro-processors, and communication buses. The resulting model of target architecture is Globally Asynchronous, Locally Synchronous (GALS). GALS architecture share some features with asynchronous ones: no global clock drives it, and no global state is available. For this and similar cases,

> the synchrony/asynchrony issue consists in providing assistance for a correct-by-construction migration from a synchronous specification to a GALS type of architecture; here, correct-by-construction means that the specified *logical* function is implemented, and some "soft" real-time *performance requirements* are satisfied.

The rest of this paper is devoted to discussing how the above three informal problems can be or have been formalized.

2 Distributed real-time control systems [6]

Here we must start from physics, since controlling a physical plant or device is, after all, the duty. Regardless of issues of digitalization and measurements, we shall consider the ideal case in which the physical system in consideration stays in closed loop with some analog feedback, and is subject to mode changes and reconfigurations. The adequate model for this is a *hybrid dynamical system.* One possible general model is the following:

$$\begin{cases} \dot{x} = f(x, v, t, q) \\ y = h(x, v, t, q) \end{cases} \tag{1}$$

$$\alpha \; = \; a(x, u) \tag{2}$$

$$\alpha \neq \alpha_- \;\; \Rightarrow \;\; T(q, \alpha, q') \tag{3}$$

Equations (1) constitute the continuous part of the model. It is a dynamical system having $t \in \mathbf{R}$ as real time, $v \in \mathbf{R}^m$ as disturbance from the environment, $x \in \mathbf{R}^n$ as continuous state, $y \in \mathbf{R}^p$ as continuous output, and $q \in Q$ belongs to some finite alphabet of discrete states, figuring the current mode — mode $q(t)$ is piecewise constant in time. Hence dynamics (1) is mode dependent. Equation (2) generates mode changes, it is driven by some control u (say, from the operator), and function a takes its values in some finite alphabet A of mode change requests. Thus α is a piecewise constant signal, which we regard as continuous from the right, i.e., $\alpha(t) = \lim_{s \searrow t} \alpha(s)$ holds, its left limits are denoted by $\alpha_-(t) := \lim_{s \nearrow t} \alpha(s)$, and a mode change request occurs when $\alpha(t) \neq \alpha_-(t)$.

This mode change request results in a mode change according to formula (3), where $T(q, \alpha, q')$ is the transition relation of a finite state automaton.

Formulas (1,2,3) can be regarded as a generic model of hybrid component, and such models can be connected by either synchronizing their mode change automata, or connecting some continuous outputs of a component to corresponding continuous inputs in another component.

Now assuming that available techniques from modern robust control theory and engineering can deal with the continuous part, let us focus on the discrete part. It consists of the two formulas (2,3), actually the key equation for our discussion is (2).

This equation summarizes the transformation, from the pair (x, u) consisting of the states and stimuli received from the environment, to the mode change request signal α. Interpret equation (3) as the discrete control implemented within the computer, then equation (2) abstracts the complex process of 1/ performing distributed measurements, digitalization, and possibly thresholding, and 2/ collecting these partial informations through the communication bus to compute change mode request. This can be summarized as in figure 1. In this

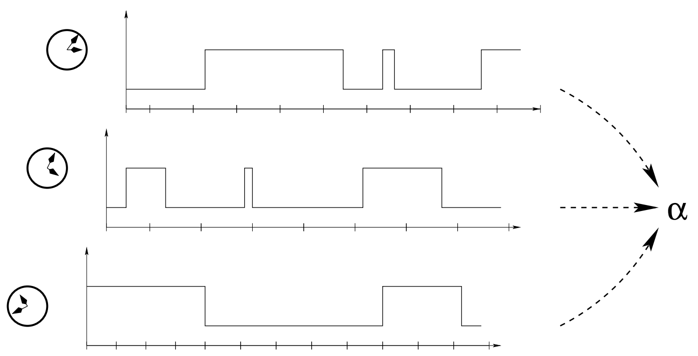

Figure 1. Distributed sensing of logical signals, and their combination.

figure, we show three different piecewise constant signals, they result from some thresholding mechanism associated to the distributed sensors. Each sensor has its own independent clock to drive A/D conversion, and the clocks are loosely synchronized (their sampling frequencies are nearly the same, but their phases are different and subject to jitter). The phases include transmission delay from sensor to computer. Clearly, for a given tuple of analog sensor records, the combination of their different *sampled* versions may differ in general, due to slight deviations occurring within each clock and nondeterminism in communication delays. Hence the computation of α can yield different results, depending on the clock drifts and jitters, and communication delays. The request is that *the behaviour of the purely discrete part* (3) *should not depend on such variations* (by this we mean that the resulting sequence of successive discrete states should remain the same). Otherwise, spurious, unexpected, behaviours of the discrete controller may arise. And these spurious behaviours would not be encompassed by the purely "synchronous" abstract model, and therefore could not be exhibited by model checking its single discrete part. The lesson is that

> robustness against uncertainties with respect to physical sampling time, communication delays, and timing uncertainties in RTOS primitives, is the very question in distributed real-time control systems.

We refer the reader to [7] for possible solutions to ensure such a robustness.

3 Synchronous hardware design from IP's [5]

To my mind, the approach I discuss here, which is due to [5], consists in adapting classical ideas from Leiserson's retiming techniques for hardware, to the more recent concept of IP. Let me briefly explain how I see this.

3.1 The folklore of retiming

A simple model of synchronous hardware consists of a data-flow graph $\mathcal{G}$, i.e., a DAG whose vertices figure variables, and branches figure dependencies. For u, v two variables, and k the integer time index:

- draw a branch $u \rightarrow v$ if v_k uses u_k for its computation, and
- draw a branch $u \xrightarrow{n} v$ if the kth occurrence, v_k, of v uses the n-shifted version u_{k-n} of u, for its computation. Call integer index n the *weight* of the branch.

Note that $u \rightarrow v$ can be seen as a short hand for $u \xrightarrow{0} v$. The weight of a path in the graph is simply the sum of the weights of its successive branches. A graph is called *well formed* if every circuit has a stricly positive weight. We consider only well formed graphs, for simplicity. Now the question is: *find modifications of $\mathcal{G}$ that will not change its semantics.* We claim that any combination of the following primitive changes has the desired property:

(a) Moving latches around. Let v be a variable. Denote by $u_1, \ldots, u_p$ the variables of $\mathcal{G}$ such that $u_i \xrightarrow{n_i} v, i = 1, \ldots, p$ are branches of $\mathcal{G}$, and denote by $w_1, \ldots, w_q$ the variables of $\mathcal{G}$ such that $v \xrightarrow{n_j} w_j, j = 1, \ldots, q$ are branches of $\mathcal{G}$. Associate to v a positive or negative integer $m \in \mathbf{Z}$. Using m, modify $\mathcal{G}$ by performing the following changes:

$$
\begin{aligned}
u_i \xrightarrow{n_i} v &\longmapsto u_i \xrightarrow{n_i - m} v \\
v \xrightarrow{n_j} w_j &\longmapsto v \xrightarrow{n_j + m} w_j
\end{aligned}
\tag{4}
$$

meaning that we pick m latches from each ingoing branch of v, and move them to each outgoing branch of it. Clearly, this shifts v by m ticks, but this does not change the map

$$(u_i, i = 1, \ldots, p) \mapsto (w_j, j = 1, \ldots, q)$$

We consider that shifting v in time is acceptable, as long as its sequence of successive values remains the same. Hence we regard transformation (4) as a semantic preserving transformation. Of course, for (4) to be feasible, we need that m satisfies the following condition (there are no "negative latches"):

$$\forall i = 1, \ldots, p \,:\, n_i - m \geq 0 \text{ and } \forall j = 1, \ldots, q \,:\, n_j + m \geq 0\,. \qquad (5)$$

One way of satisfying condition (5) is to provide input signals with additional latches. This only increases the overall latency of the circuit, but otherwise does not change its behaviour. If enough input latches are added, then satisfying (5) is eventually possible.

(b) Upsampling. The successive occurrences of variable v are denoted by $v_k, k = 1, 2, \ldots$ Now, pick $J > 1$ an integer, and set

$$v'_l := \textbf{ if } l = kJ \textbf{ then } v_k \textbf{ else } \tau\,,$$

where τ is a special value, called *non informative,* not belonging to any useful domain, τ is to be interpreted as a don't care. Ignoring don't cares in v' yields again v. If we perform this globally using the same integer J at all vertices of the data-flow graph $\mathcal{G}$, we preserve the semantics (just ignore don't cares, as said before).

Now, applying successively transformations of types (a) or (b) in any order, still preserves the semantics: the different signals are preserved, up to a shift, and ignoring don't cares yields again the original signals. Playing with this allows to adjust throughput and latency of the circuit. This is classical stuff, very useful indeed to synthesize pipelining.

Note that transformation (b) can be generalized by allowing a "variable upsampling" : there is no need that informative values in a signal be separated by a fixed number J of don't cares. The only thing which matters is that don't cares should be inserted simultaneously, for every variable. Then ignoring don't cares yields again the original signals. Therefore, the upsampling can be, so to say, modified "on-line" while the circuit is running.

Well, assume we wish to implement the function specified by the original $\mathcal{G}$ (I mean prior to any transformation (a) or (b)), in the form of a hardware block being part of a larger circuit. Then, as argued in section 1, input wires may be shifted in time, due to "interconnect-delay problem". If this shift is fixed, then we can use transformations (a) and (b) to compensate for such a mismatch. However, it may be the case that this interconnect-delay can vary with time, just because routing can be data dependent, dynamically. Then, we need to resource to the generalized rule (b) to solve the problem.

This is fine if we have the environment of the considered block at hand. But we are still not done if such an information is not available. Unfortunately, this is precisely the case for IP's, since they are designed prior to their possible reuse, so their future environments are not known! The nice idea of [5] is *to design a priori the considered block in such a way that it adjusts automatically to any interconnect-delay, on-line.* This is called latency insensitive design, we briefly explain it hereafter.

3.2 The models used

There are basically two models of interest:

strictly synchronous : According to this model, there is some given finite set V of variables, and a *state* x assigns an effective value to each variable $v \in V$. A *strictly synchronous behaviour* is a sequence $s = x_1, x_2, \ldots$ of states. A *strictly synchronous process* is a set of strictly synchronous behaviours. A *strictly synchronous signal* is the sequence of values $s_v = v(x_1), v(x_2), \ldots,$ for $v \in V$ given. Hence all signals are indexed by the same totally ordered set of integers $\mathbf{N} = \{1, 2, \ldots\}$ (or some finite prefix of it). Hence all behaviours are *synchronous* and are tagged by the same clock, this is why I use the term "strictly" synchronous. In practice, strictly synchronous processes are specified using a set of legal *strictly synchronous reactions* R, and therefore strictly synchronous processes take the form

$$P \;=\; R^{\omega} ,$$

where superscript "$.^{\omega}$" denotes unbounded iteration. Composition is defined as the intersection of the set of behaviours, it is performed by taking the conjunction of reactions:

$$P \,\|\, P' \;:=\; P \cap P' = (R \wedge R')^{\omega} . \tag{6}$$

This is the classical mathematical framework used in synchronous hardware modelling. It raises problems when using it as a framework for IP design, as it does not comply with retiming.

synchronous : Here the model is the same as in the previous case, but every domain of data is enlarged with some *non-informative* value, denoted by the special symbol τ. A τ value is to be interpreted as a don't care. Values different from τ are called *informative,* since they correspond to actual, useful values for use in computing the outputs of the circuit. Besides this, things are as before: a *state* x assigns an informative or non-informative value to each state variable $v \in V$, where V is some given finite set of variables. A *synchronous behaviour* is a sequence of states. A *synchronous process* is a set of synchronous behaviours. A *synchronous signal* is the sequence of informative or non-informative values $s_v = v(x_1), v(x_2), \ldots,$ for $v \in V$ given. And composition is performed as in (6). Hence, strictly synchronous processes are just synchronous processes involving only informative values.

3.3 The problems

The idea is that we wish to implement a strictly synchronous specification P by means of a synchronous process P^{ℓ}, insensitive to latency. Then P^{ℓ} is the version of P for use as an IP block. For this the following problems need to be addressed.

Problem 1 : how to model that a synchronous process P^ℓ implements a strictly synchronous specification P, while being insensitive to latency ? The picture is that valuations of variables travel on wires of the design, and this causes latency and may require pipelining. Such latency may take several clock cycles, and can differ for different variables (since different wires are used). How to reflect this in the model ? For $v \in V$, pick some signal

$$s_v = v(x_1), v(x_2), v(x_3), \ldots \in P\,. \tag{7}$$

To reflect a wire-dependent latency, the same signal, observed later on along a wire, has (for example) the form

$$s_v^\ell = \tau, v(x_1), \tau, v(x_2), \tau, \tau, v(x_3), \ldots \tag{8}$$

i.e., τ symbols can be inserted at arbitrary places of the original signal s_v. This is the mechanism of *stalling* a signal. Conversely, for s_v^ℓ as in (8), its *strict* counterpart is given in (7). For v a variable, denote by

$$\chi_v : s_v^\ell \longmapsto s_v$$

the map giving the strict version of a stalled signal, and denote by χ_V the tuple composed of the χ_v where v ranges over the considered set V of variables. For P^ℓ a process having V as set of variables, its image $\chi_V(P^\ell)$ by χ_V is called the *strict process* corresponding to P^ℓ.

A process P^ℓ is called *patient* iff it satisfies the following : for all $s \in P^\ell$, all input signal s_i of s, and all instant k, there exists another behaviour $stall(s) \in P^\ell$, whose i-signal coincides with s_i before instant k, has a τ-event at k, and can be further stalled after k. It is clear (but tedious verifying) that the i/o-connection of patient processes is patient. The intuition is that a patient process can still offer the desired reaction to an input event if this input is delayed.

Let us formalize this some more. A *singleton buffer* is any process which possesses two variables v_i, v_o (standing for "input" and "output"), and has the identity process $s_{v_o} := s_{v_i}$ as corresponding strict process [2]. A *buffer* is simply the parallel composition of finitely many singleton buffers involving disjoint sets of variables (i.e., having no interaction). It has V_i, V_o, two copies of the same set V, as input and output variables ; for each $v \in V$, the pair (v_i, v_o) is related via a singleton buffer. In the following theorem, P^ℓ and Q^ℓ are two processes having disjoint sets of variables, communicating by means of a buffer having V_i, V_o as sets of input and output variables (V_i is shared by P^ℓ and the buffer, and V_o is shared by the buffer and P^ℓ).

Theorem 1 ([5]). *If P^ℓ and Q^ℓ are* patient *processes, and B, B' are two buffers as stated before, then*

$$\chi_V(P^\ell \parallel B \parallel Q^\ell) = \chi_V(P^\ell \parallel B' \parallel Q^\ell) \;\;=\;\; P \parallel Q\,,$$

where P, Q are the strict processes corresponding to P^ℓ and Q^ℓ.

[2] Here I simplify by accepting unbounded buffers. A more precise analysis is performed in [5].

Setting for instance $B' = Id$ (the identity buffer), theorem 1 implies in particular that inserting a buffer does not change the corresponding strict process.

Problem 2 : for $\boldsymbol{P}$ a strict process, how to build a *patient* process $\boldsymbol{P^\ell}$ such that $\boldsymbol{\chi_V(P^\ell) = P}$? Here I deviate from [5] and rather propose a solution formulated as an extension of the classical technique of subsection 3.1. The idea is to interleave communication rounds and adaptive retiming, and to perform actual computations only when corresponding functions are provided with informative values for all their input variables. We detail this next.

Our starting point is the data-flow graph $\mathcal{G}$ serving as our original specification. Hence $\mathcal{G}$ implements a strict process and is supposed to process only informative values, it is defined as in subsection 3.1. To encompass the interaction with the environment, we enlarge $\mathcal{G}$ with additional branches of the form

$$\longrightarrow u \tag{9}$$

where $u \in V^{\mathrm{i}}$ (the set of input variables), and the missing vertex at the origin of the branch refers to the environment.

To each variable v of $\mathcal{G}$ we associate a weight as in (4), we denote it by m_v. The tuple of all m_v, for v ranging over V, is denoted by m_V. The moving of latches is equivalently encoded by our set of weights m_V, and we must restrict ourselves to sets of weights satisfying the feasibility condition (5). As an initial value for m_V we set

$$\forall v \in V : \; m_v := 0 \,. \tag{10}$$

Hence our original data structure to model the circuit is the pair $(\mathcal{G}, \mathbf{0})$ consisting of $\mathcal{G}$ together with the zero weighting (10). Then the pair $(\mathcal{G}, m_V)$ is updated on-line, at each reaction, according to the following protocol.

As a simple case, suppose that all input wires of $\mathcal{G}$ receive informative values for the first round. Then the reaction can proceed as specificed by $\mathcal{G}$ directly, and the circuit can wait for a second set of input values. This was the trivial case. The interesting case is of course when at least one input wire offers a non-informative value τ for the first reaction. Assume this occurs exactly for one single $u \in V^{\mathrm{i}}$. What can we do ?

The key idea is to model the reception of a non informative value on input wire u via the insertion of a *negative* delay in the corresponding input branch of $\mathcal{G}$:

$$\text{update } \mathcal{G} \;:\; [\xrightarrow{0} u\,] \longmapsto [\xrightarrow{-1} u\,] \tag{11}$$

Of course this negative delay can be equivalently implemented by buffering all informative values received at the other input wires, for proper resynchronization at the next reaction with the considered late input wire u.

But we can be more clever indeed. Denote by

$$u^{\rightarrow}$$

the subset of the variables $v \in V$ which instantaneously depend on input u, i.e., there exists a path from u to v in $\mathcal{G}$ having zero weight. Then we update the set of weights m_V as follows:

$$\begin{array}{lcl} \forall v \in u^{\rightarrow} & : & m_v := m_v - 1 \\ \forall v \notin u^{\rightarrow} & : & m_v := m_v \end{array} \tag{12}$$

Using the weights (12) we apply rule (4) of retiming. Having $m_v = -1$ for $v \in u^{\rightarrow}$ results in: 1/ assuming the availability of one latch at the output wires belonging to $u^{\rightarrow}$, and, 2/ moving these latches backward until a variable not belonging to $u^{\rightarrow}$ is reached. In passing we have compensated the negative delay in front of u by a positive one, therefore making the whole synchronization correct. As a result, variables not belonging to $u^{\rightarrow}$ are evaluated immediately, and the evaluation of other variables is delayed, hence τ values are not effectively processed. This results in an eager evaluation scheme implementing the original, strict, specification.

Generalizing this idea results in the following protocol for updating on-line the pair $(\mathcal{G}, m_V)$ at each reaction, where notation $s_u(x)$ denotes the value of input variable u for the considered behaviour, at the current reaction:

$$\text{update } \mathcal{G} : \;\; s_u(x) = \tau \;\Rightarrow\; [\xrightarrow{n} u] \longmapsto [\xrightarrow{n-1} u] \tag{13}$$

$$\text{update } m_V : \begin{cases} \forall v \in U_\tau^{\rightarrow} \;\; : \;\; m_v := m_v - 1 \\ \forall v \notin U_\tau^{\rightarrow} \;\; : \;\; m_v := m_v \\ \quad\;\; \text{where } \;:\; U_\tau^{\rightarrow} = \bigcup_{u\,:\,s_u(x)=\tau} u^{\rightarrow} \end{cases} \tag{14}$$

One particular case of interest for this protocol is that of a *stuttering* or *stalling* reaction, in which all input wires receive a non informative value. Then a negative delay is added at each input wire, which is compensated by backward propagating, from each output wire, a (positive) delay. Of course we better do nothing and simply skip this reaction! In general, this protocol is applied by suitable adaptive routing of the data through a predefined set of latches. This can be done at the chosen level of granularity (here I assume the finest level, namely one node of the graph $\mathcal{G}$ per function, but *one node per IP* is probably a more realistic choice, further analyzed in [5]).

Comparing this solution to the one proposed in [5], our adaptive retiming protocol corresponds to the *equalizers* of [5], and the use of stuttering or stalling reactions correspond to the *stallable* processes of [5]. The latter study is more precise, however, since it takes into account the request of resourcing only to a bounded set of latches.

4 Building software or hardware GALS architectures [2][3][4]

Here we discuss a different situation. We wish to assist the mapping from synchronous specifications to GALS implementations. Here we insist on *logical* correctness, meaning that each synchronous component should have exactly the *same* set of behaviours, be it composed synchronously (as in the specification) or *a*synchronously with the other synchronous components (as in the GALS implementation architecture). As said before, logical correctness is the driving requirement, real-time requirements are soft ones, and are generally ancillary.

Thus the requirements share some similarity with those of section 3. But there are significant differences: our GALS model is not "globally ticked", and does not have a global state. The problem considered in this section is in fact an extension of the one considered in section 3.

4.1 The models used

Here we consider two *different* models:

synchronous : Processes progress via an infinite sequence of *reactions*:

$$P \;=\; R^\omega$$

where R denotes the family of possible *reactions,* a reaction is a transition relation relating successive states. Within a state, some events can be *absent,* modelled by the occurrence of a special, non informative symbol τ, also denoted by $\perp$ following the notations of [2][3][4]. And the control can use the absence of these events as a viable information. Parallel composition is given by taking the pairwise conjunction of associated reactions, whenever they are composable:

$$P_1 \| P_2 \;=\; (R_1 \wedge R_2)^\omega$$

This results in having

$$P_1 \| P_2 = P_1 \cap P_2$$

This model coincides with the synchronous model of section 3.

***a*synchronous** : Reactions cannot be observed any more, no global clock exists. Instead a *behaviour* is a tuple of *signals,* and each individual signal is a totally ordered sequence of (informative) events. A process P^a is a set of behaviours. "Absence" cannot be sensed, and has therefore no meaning. Composition occurs by means of unifying each individual signal shared between two processes:

$$P_1^a \|_a P_2^a := P_1^a \cap P_2^a$$

This models in particular the communications via asynchronous unbounded FIFOs. More generally, a reliable communication medium is assumed, in which messages are not lost, and for each individual channel, messages are sent and received in the same order, but no global synchronization is assumed.

In our study we consider in particular the following two different architectures:

synchronous architecture : see above. We use it for the specification stage.

GALS architecture : a network of synchronous processes, interconnected by asynchronous communications (as defined above). We use it as deployment architecture.

And the central issue is: *what do we preserve when deploying a synchronous specification on a* GALS *architecture?*

4.2 The fundamental problems

Before stating the problems, let me say that [2][3][4] assumes that the considered synchronous processes are *stuttering invariant,* meaning that in each state x it is always possible to keep "silent", by not changing the state and not emitting anything (i.e. emitted variables have the value $\perp$). Stuttering invariance is a key notion when considering open systems, since it is always possible, for a system, to be subsequently combined with another one which will be active while the considered one stutters. Note that suttering invariance has some similarity with "stallability" introduced in [5] and discussed in section 4.1.

Problem 1: what if a synchronous block receives its data from an asynchronous environment? Focus on a synchronous block within a GALS architecture, it receives its inputs as a tuple of (non synchronized) signals. Since some events can be absent in a given state, it can be the case that some signal will not be involved in a given reaction. But since the environment is asynchronous, this information is not provided by the environment. In other words, the environment does not offer to the synchronous block the correct model for its input stimuli. In general this will drastically affect the semantics of the block. However, some particular synchronous blocks are robust against this type of difficulty. How to formalize this?

Let P be such a block. Let $\sigma = x_0, x_1, x_2, \ldots$ be a behaviour of P, i.e., a sequence of states compliant with the reactions of P. Let V be the (finite) set of state variables of P, each state is a valuation of all $v \in V$, including the possible special status *absent,* which is denoted by the special symbol $\perp$ in [2][3][4], the valuation of v at state x is written $v(x)$. Hence we can write equivalently

$$\begin{aligned}\sigma &= (v(x_0))_{v\in V}, (v(x_1))_{v\in V}, (v(x_2))_{v\in V}, \ldots \\ &= (v(x_0), v(x_1), v(x_2), \ldots)_{v\in V} \\ &=_{\text{def}} (\sigma_v)_{v\in V}\end{aligned}$$

Now, for each separate v, remove the $\perp$ from the sequence $\sigma_v = v(x_0), v(x_1), v(x_2), \ldots$, this yields a (strict) *signal*

$$s_v \;=_{\text{def}}\; s_v(0), s_v(1), s_v(2), \ldots$$

where $s_v(0)$ is the first non $\perp$ term in σ_v and so on. Finally we set

$$\sigma^a \;=_{\text{def}}\; (s_v)_{v\in V}$$

The so defined map $\sigma \mapsto \sigma^a$ takes a synchronous behaviour, and returns a uniquely defined asynchronous one. This results in a map

$$P \longmapsto P^a$$

defining the *desynchronization* P^a, of P. Clearly, the map $\sigma \mapsto \sigma^a$ is not one-to-one, and thus it is not invertible. However, we have shown in [2][3][4] the first fundamental result that

$$\begin{array}{l}\text{if } P \text{ satisfies a special condition called } \textit{endochrony,} \text{ then} \\ \forall \sigma^a \in P^a \text{ there exists a unique } \sigma \in P \text{ such that } \sigma \mapsto \sigma^a \text{ holds!}\end{array} \quad (15)$$

This means that, by knowing the formula defining reaction R such that $P = R^\omega$, we can uniquely reconstruct a synchronous behaviour, from observing its desynchronized version. In addition, it is shown in [2][3][4] that this reconstruction can be perfomed *on-line* meaning that each continuation of a prefix of σ^a yields a corresponding continuation for the corresponding prefix of σ. The important point about result (15) is that endochrony can be model checked on the reaction R defining the synchronous process P. Also,

$$\text{any } P \text{ can be given a } \textit{wrapper } W \text{ making } P \| W \text{ endochronous.} \quad (16)$$

How can we use (15) to solve problem 1 ? Let E be the model of the environment, it is an asynchronous process according to our above definition. Hence we need to formalize what it means having "P interacting with E" since they do not belong to the same world. The only possible formal meaning is

$$P^a \parallel_a E$$

Hence having P^a interacting with E results in an asynchronous behaviour $\sigma^a \in P^a$, but using (15) we can reconstruct uniquely its synchronous counterpart $\sigma \in P$. So this solves problem 1.

However considering problem 1 is not enough, since it only deals with a single synchronous block interacting with its asynchronous environment, it does not consider the problem of mapping a synchronous network of synchronous blocks onto a GALS architecture.

Problem 2 : what if we deploy a synchronous network of synchronous blocks onto a GALS architecture ? Consider the simple case of a network of two blocks P and Q. Since our communication media behave like a set of FIFOs, one per each signal sent from one block to the other, we perfectly know what the desynchronized behaviours of our deployed system will be:

$$P^a \parallel_a Q^a$$

There is not need for inserting any particular explicit model for the communication medium, since by definition $\|_a$-communication preserve each individual asynchronous signal (but not their global synchronization!). In fact, Q^a will be the asynchronous environment for P^a and vice-versa.

Now, if P is endochronous, then, by having solved problem 1 we can uniquely recover a synchronous behaviour σ for P, from observing an asynchronous behaviour σ^a for P^a as produced by $P^a \parallel_a Q^a$.

But we are still not happy: it can be the case that there exist some asynchronous behaviour σ^a for P^a produced by $P^a \parallel_a Q^a$, which *cannot be obtained* by desynchronizing the synchronous behaviours of $P \parallel Q$. In fact we only know in general that

$$(P \parallel Q)^a \subseteq (P^a \parallel_a Q^a) \tag{17}$$

However, we have shown in [2][3][4] the second fundamental result that

$$\begin{array}{l}\text{if } (P,Q) \text{ satisfies a special condition called } \textit{isochrony,} \\ \text{then equality in (17) indeed holds!}\end{array} \tag{18}$$

The nice thing about isochrony is that it is *compositional*: if P_1, P_2, P_3 are pairwise isochronous, then $((P_1 \| P_2), P_3)$ is an isochronous pair, so we can refer to an *isochronous network* of synchronous processes. Also, isochrony enjoys additional useful properties listed in [2][3][4]. Again, the condition of isochrony can be model checked on the pair of reactions associated to the pair (P,Q), and

$$\begin{array}{l}\text{any pair } (P,Q) \text{ can be given } \textit{wrappers } (W_P, W_Q) \\ \text{making } (P \| W_P, Q \| W_Q) \text{ an isochronous pair.}\end{array} \tag{19}$$

Just a few additional words about the condition of isochrony, which is of interest per se. Synchronous composition $P \| Q$ is achieved by considering the conjunction

$$R_P \wedge R_Q$$

of corresponding reactions. In taking this conjunction of relations, we ask in particular that common variables have identical status present/absent in both components, in the considered reaction. Assume we relax this latter requirement by simply asking that the two reactions should only agree on effective values of common variables, *when they are both present.* This means that a given variable can be freely present in one component but absent in the other. This defines a "weakly synchronous" conjunction of reactions, we denote it by

$$R_P \wedge_a R_Q$$

In general, $R_P \wedge_a R_Q$ has more legal reactions than $R_P \wedge R_Q$. It turns out that the isochrony condition for the pair (P,Q) writes:

$$(R_P \wedge R_Q) \equiv (R_P \wedge_a R_Q)$$

4.3 A sketch of the resulting methodology

How can we use (15) and (18) for a correct deployment on a GALS architecture? Well, consider a synchronous network of synchronous processes

$$P_1 \parallel P_2 \parallel \ldots \parallel P_K$$

such that

(GALS_1) ***: each P_k is endochronous, and***
(GALS_2) ***: the $P_k, k = 1, \ldots, K$ form an isochronous network.***

Using condition (GALS_2), we get

$$P_1^a \parallel_a (P_2^a \parallel_a \ldots \parallel_a P_K^a) \;=\; (P_1 \parallel P_2 \parallel \ldots \parallel P_K)^a$$

Hence every asynchronous behaviour σ_1^a of P_1^a produced by its interaction with the rest of the asynchronous network $(P_2^a \parallel_a \ldots \parallel_a P_K^a)$ is a desynchronized version of a synchronous behaviour of P_1 produced by its interaction with the rest of the synchronous network. Hence the asynchronous communication does not add spurious asynchronous behaviour. Next, by (GALS_1), we can reconstruct on-line this unique synchronous behaviour σ_1, from σ_1^a. Hence,

Theorem 2. *For $P_1 \parallel P_2 \parallel \ldots \parallel P_K$ a synchronous network, assume the deployment is simply performed by using an asynchronous mode of communication between the different blocks. If the network satisfies conditions* (GALS_1) *and* (GALS_2)*, then the original synchronous semantics of each individual block of the deployed* GALS *architecture is preserved (of course the global synchronous semantics is not preserved).*

To summarize, *a synchronous network satisfying conditions* (GALS_1) *and* (GALS_2) *is the right model for a* GALS*–targetable design,* and we have a correct-by-construction deployment technique for GALS architectures. The method consists in preparing the design to satisfy (GALS_1) and (GALS_2) by adding the proper wrappers, and then performing bruteforce desynchronization as stated in theorem 2.

5 Discussion

Of course there are similarities :

- The general concern is the same, namely to formalize the construction of modular architectures by providing a correct by construction methodology.
- The role and use, in sections 3 and 4, of special symbols e.g., $\perp$ and τ is similar [3]. Stuttering invariant processes of [2][3][4] very much correspond to stallable processes of [5]. The mechanism of stalling in [5] is reminiscent of the

[3] The use of "absent" or "non-informative" $\perp$ symbol was considered important, from the very beginning, when developing the SIGNAL synchronous language [1][9].

desynchronization mechanism of [2][3][4] (τ's are wildly inserted in individual signals in the former case, whereas synchronization barriers of reactions are lost in the latter). Also, system distribution in real-time control systems (section 2) also causes nondeterministic delay.
- The use of equalizers in [5] resembles the special protocol used in [2][3][4] to reconstruct σ from σ^a, see the detailed papers. Both involve a certain type of "blocking" or "delaying" reactions until necessary information is made available. It turns out that using "thick" events in combination with the so-called *confirmation* mechanism proposed by Caspi, performs the same for distributed real-time control systems, but in an implicit way.

Now there are differences :

- The work of [2][3][4] was intended to target architectures in which *no global clock nor state* is available, this differs from [5] in which synchronized hardware was considered, therefore providing global clock and state. Worse, for distributed real-time control systems, handling discrete events is not enough and the problem does not reduce to the only preserving of the discrete time/discrete event semantics.
- The approach of [5] is suited to synchronized hardware, it is not applicable to distributed software architectures in which communication is offered by some OS or middleware services. In [2][3][4] we were rather targeting either pure SW or mixed HW/SW. For HW, the techniques of [2][3][4] still apply, a variation of it has been used by the SIGNAL team with Paul Le Guernic and colleagues [8].
- In [5], the early spec for an IP is in fact the strict process it implements, there every signal has only informative events and an identical clock. In [2][3][4], the early spec is "multiclock" from the beginning, we distinguish between "running signals" and, say, "interruptions" and consider they are tagged by *different* clocks — they are not synchronous in the sense of [5]. Again, the early spec for distributed real-time control systems is much more subtle, as it already involves *robustness requirements,* from the very beginning, against variations of phase and delay in the {system + controller} dynamics, but these robustness requirements are considered mainly for the continuous control part. Then the very question is to make sure that distributing the computer control architecture still guarantees that the above robustness is maintained, including the discrete control part.

To conclude, "think synchronously – act asynchronously" emerges as a common paradigm for the design of embedded systems from components. We have discussed three different cases implementing it (the case of real-time control is still currently under finalization by Paul Caspi & friends). In some sense, one can see deep similarities between these different cases, and I can imagine that knowing the techniques used therein would allow one to find the proper adaptation for yet another, slightly different situation. Topologists or dynamicists are challenged to understand, capture, and formulate these diverse situations within a common mathematical framework, this challenge is certainly non trivial, given the diversity of the different mathematical approaches today.

ACKNOWLEDGEMENTS. *I am indebted to many people, who have gently stimulated my quest for the truth. Paul Caspi has consistently warned me that the desynchronization techniques of section 4 would not yield sensible results for distributed* real-time *control systems, although importing these techniques was formally feasible; he convinced me just-in-time for the preparation of this paper. I thought synchronization would not be an issue for systems equipped with a global clock, and I am grateful to Luca Carloni, Dave Mc Millan, and Alberto Sangiovanni Vincentelli for demonstrating the converse to me.*

References

1. A. Benveniste and P. Le Guernic. "Hybrid dynamical systems theory and the Signal language." *IEEE Trans. on Automatic Control,* AC-35(5): 535-546, 1990.
2. A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, 163, 125-171 (2000).
3. A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99, Concurrency Theory, 10th International Conference*, volume 1664 of *Lecture Notes in Computer Science*, pages 162–177. Springer, August 1999.
4. A. Benveniste, B. Caillaud, P. Le Guernic, and J.P. Talpin. Desynchronization of synchronous programs: summary of results. Preprint, available at http://www.irisa.fr/sigma2/benveniste/pub/B_al99.html
5. L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. The theory of latency insensitive design. Submitted for publication, 2001.
6. P. Caspi and R. Salem. Threshold and Bounded-Delay Voting in Critical Control Systems. *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems*, September 2000, Joseph Mathai Ed., 68–81, Lecture Notes in Computer Science, vol 1926.
7. P. Caspi. Embedded control: from asynchrony to synchrony and back. This volume.
8. A. Kountouris and C. Wolinski. A method for the generation of HDL code at the RTL level from a high-level formal specification language. In *Proc. of MWSCAS'97,* IEEE Computer Society Press, Sacramento, Aug. 1997.
9. P. Le Guernic and T. Gautier. Data-flow to von Neumann: the SIGNAL approach. In *Advanced topics in data-flow computing,* J-L. Gaudiot, L. Bic Eds., 413–438, Prentice Hall, 1991.
10. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87,* Vol. 22 of ACM SIGPLAN Notices, ACM Press, 1987.
11. J. McAffer. Meta-Level Architecture Support for Distributed Objects. In *Proceedings of Reflection 96,* G. Kiczales (ed), San Francisco, USA, March 1996.
12. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, Optimized Sun RPC using Automatic Program Specialization. In *Proceedings ICDCS'98,* Amsterdam, May 1998.
13. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings 15th ACM Symposium on Operating Systems Principles,* Copper Mountain Resort, Colorado, USA, December 1995.