Non-massive, non-high performance, distributed computing: selected issues*

Albert Benveniste[1]

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

Abstract There are important distributed computing systems which are neither massive nor high performance. Examples are: telecommunications systems, transportation or power networks, embedded control systems (such as embedded electronics in automobiles), or Systems on a Chip. Many of them are *embedded systems,* i.e., not directly visible to the user. For these systems, performance is not a primary issue, major issues are reviewed in this paper. Then, we focus on a particular but important point, namely the correct implementation of specifications on distributed architectures.

1 Beware

This is a special and slightly provocative section, just to insist, for the Euro-Par community, that:

> ***there are important distributed computing systems which are neither massive nor high performance.***

Here is a list, to mention just a few:

(a) Telecommunications or web systems.
(b) Transportation or power networks (train, air-traffic management, electricity supply, military command and control, etc.).
(c) Industrial plants (power, chemical, etc.).
(d) Manufacturing systems.
(e) Embedded control systems (automobiles, aircrafts, etc.).
(f) System on Chip (SoC) such as encountered in consumer electronics, and Intellectual Property (IP)-based hardware.

Examples (a,b) are distributed, so to say, by tautology: they are distributed because they are networked. Examples (c,d,e) are distributed by requirement from the physics: the underlying physical system is made of components, each component is computerized, and the components concur at the overall behaviour of the system. Finally, example (f) is distributed by requirement from the electrons: billion-transistor SoC cannot be globally synchronous.

* This work is or has been supported in part by the following projects : Esprit R&D SAFEAIR, and Esprit NoE ARTIST.

Now, (almost) all the above examples have one fundamental feature: *they are open systems, which interact continuously with some unspecified environment having its own dynamics.* Furthermore, some of these open systems interact with their environment in a tight way, e.g. (c,d,e) and possibly also (f). These we call

reactive systems,

which will be the focus of this paper. For many reactive systems, computing performance is not the main issue. The extreme case is avionics system, in which the computing system is largely oversized in performance. Major requirements, instead, are [20]:

Correctness: *the system should behave the way it is supposed to.*

Since the computer system interacts with some physical system, we are interested in the resulting closed-loop behaviour, i.e., the joint behaviour of the physical plant and its computer control system. Thus, specifying the signal/data processing and control functionalities to be implemented is a first difficulty, and sometimes even a challenge (think of a flight control system for a modern flight-by-wire aircraft). Extensive virtual prototyping using tools from scientific and control engineering is performed to this end, by using typically Matlab/Simulink with its toolboxes.

Another difficulty is that such reactive systems involve many *modes* of operation (a mode of operation is the combination of a subset of the available functionalities). For example, consider a modern car equipped with computer assisted emergency breaking. If the driver suddendly strongly brakes, then the resulting strong increase in the brake pedal pressure is detected. This causes the fuel injection mode to stop, ABS mode to start, and the maximal braking force is computed on-line and applied automatically, in combination with ABS. Thus mode changes are driven by the pilot, they can also be driven automatically, being indirect consequences of human requests, or due to protection actions.

There are many such modes, some of them can run concurrently, and their combination can yield thousands to million of discrete states. This discrete part of the system interfers with the "continuous" functionalities in a bidirectional way: the monitoring of continuous measurements triggers protection actions, which results in mode changes; symmetrically, continuous functionalities are typically attached to modes. The overall system is called *hybrid,* since it tightly combines both continuous and discrete aspects. This discrete part, and its interaction with the continuous part, is extremely error prone, its correctness is a major concern for the designer.

For some of these systems, real-time is one important aspect. It can be *soft* real-time, where requested time-bounds and throughput are loose, or *hard* real-time, where they are strict and critical. This is different from requesting high performance in terms of average throughput.

As correctness is a major component of safety, it is also critical that the actual *distributed implementation*—also called ***distributed deployment*** in the sequel—of the specified functionalities and mode changes shall be performed in

a correct way. After all, the implementation matters, not the spec! But the implementation adds a lot of nondeterminism: RTOS (real-time operating system), buses, and sometimes even analog-to-digital and digital-to-analog conversions. Thus a careless deployment can impair an otherwise correct design, even if the computer equipment is oversized.

Robustness: *the system should resist to (some amount of) uncertainty or error.*

No real physical system can be exactly modeled. Models of different accuracies and complexities are used, for the different phases of the scientific engineering part of the systems design. Accurate models are used for mechanics, aerodynamics, chemical dynamics, etc., when virtual simulation models are developed. Control design uses simple models reflecting only some facets of the systems dynamics. The design of the discrete part for mode switching usually oversimplifies the physics. Therefore, the design of all functionalities, both continuous and discrete, must be robust against uncertainties and approximations in the physics. This is routine for the continuous control engineer, but still requires modern control design techniques. Performing this for the discrete part, however, is still an open challenge today.

Fault-tolerance is another component of robustness of the overall system. Faults can occur, due to failures of physical components. They can be due to the on-board computer and communication hardware. They can also originate from residual faults in the embedded software. *Distributed architectures are a key counter-measure against possible faults*: separation of computers helps mastering the propagation of errors. Now, special principles should be followed when designing the corresponding distributed architecture, so as to limit the propagation of errors, not to increase its risk! For example, rendez-vous communication may be dangerous: a component failing to communicate will block the overall system.

Scope of this paper: Addressing all the above challenges is certainly beyond a single paper, and even more beyond my own capacity. I shall restrict myself to examples (e,f), and to a lesser extend (c,d). There, I shall mainly focus on the issue of correctness, and only express some considerations related to robustness. Moreover, since the correctness issue is very large, I shall focus on the *correctness of the distributed deployment, for so-called embedded systems.*

2 Correct deployment of distributed embedded applications

As a motivating application example, the reader should think of safety critical embedded systems such as flight control systems in flight-by-wire avionics, or anti-skidding and anti-collision equipment in automobiles. Such systems can be characterized as *moderately distributed,* meaning that:

– The considered system has a “limited scope”, in contrast with large distributed systems such as telecommunication or web systems.

– All its (main) components *interact,* as they concur at the overall correct behaviour of the system. Therefore, unlike for large distributed systems, the aim is not that different services or components should not interact, but rather that they should interact in a correct way.
– *Correctness,* of the components and of their interactions with each other and with the physical plant, is critical. This requires tight control of synchronization and timing.
– The design of such systems involves methods and tools from the underlying technical engineering area, e.g., mechanics and mechatronics, control, signal processing, etc. *Concurrency* is a natural paradigm for the systems engineer, not something difficult to be afraid of. The different functionalities run by the computer system operate concurrently, and they are concurrent with the physical plant.
– For systems architecture reasons, not performance reasons, deployment is performed on distributed architectures. The system is distributed, and even some components themselves can be distributed—they can involve intelligent sensors & actuators, and have part of their supervision functionalities embedded in some centralized computer.

Methods and tools used, and corresponding communication paradigms: The methods and tools used are discussed in Fig. 1. In this figure, we show on the left the

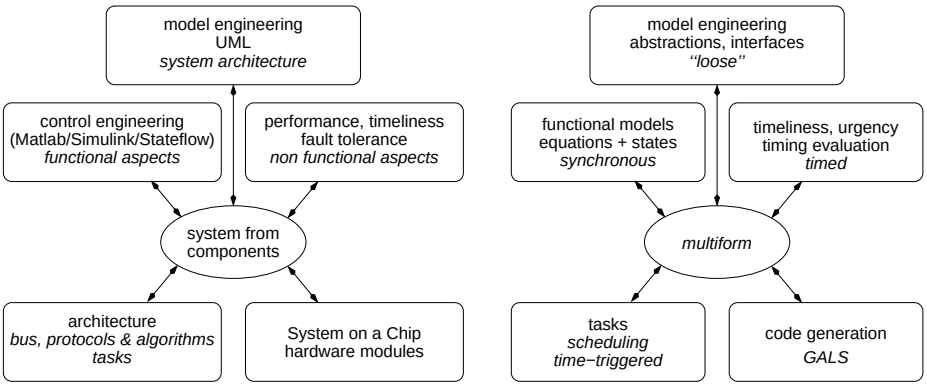

Figure1. Embedded systems: overview of methods and tools used (left), and corresponding communication paradigms (right). The top row ("model engineering") refers to the high level system specification, the second row ("control engineering") refers to the the detailed specification of the different components (e.g., anti-skidding control subsystem). And the bottom row refers to the the (distributed) implementation.

different tool-sets used throughout the systems design. This diagram is mirrored on the right hand side of the same figure, where the corresponding communication paradigms are shown.

Let us focus on the functional aspects first. This is a phase of the design in which scientific engineering tools (such as the Matlab family) are mainly used, for functionalities definition and prototyping. In this framework, there is a natural global time available. Physical continuous time triggers the models developed at the functionalities prototyping phase, in which controllers interact with a physical model of the plant. The digital controllers themselves are discrete time, and refer to some unique global discrete time. Sharing a global discrete time means using a *perfectly synchronous* communication paradigm, this is indicated in the diagram sitting on the right.

Now, some parts of the system are (hard or soft) real-time, meaning that the data handled are needed and are valid only within some specified window of time: buffering an unbounded amount of data, or buffering data for unbounded time, is not possible.

For these first two aspects, tight logical or timed synchronization is essential.

However, when dealing with higher level, global, systems architecture aspects, it may sometimes happen that no precise model for the the components interaction is considered. In this case the communication paradigm is left mostly unspecified. This is a typical situation within the UML (Universal Modeling Language) [19] community of systems engineering.

Focus now on the bottom part of this figure, in which deployment is considered. Of course, there is no such thing like a "loose" communication paradigm, but still different paradigms are mixed. Tasks can be run concurrently or can be scheduled, and scheduling may or may not be based on physical time. Hybrid paradigms are also encountered within Systems on a Chip (SoC), which typically follow a Globally Asynchronous Locally Synchronous (GALS) paradigm.

Fig. 2 shows a different view of the same landscape, by emphasizing the different scheduling paradigms. In this figure, we show a typical control structure of

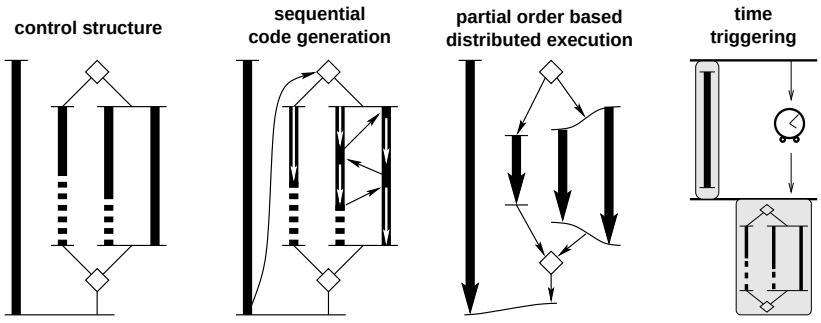

Figure2. Embedded systems: scheduling models for execution.

a functional specification (left) with its multi-threaded logical control structure. The horizontal bars figure synchronization points, the (dashed) thick lines figure (terminated) threads, and the diamonds indicate fork/joins. This functional

specification can be compiled into non-threaded sequential code by generating a total order for the threads (mid-left), this has the advantage of producing deterministic executable code for embedding.

But a concurrent, and possibly distributed, execution is also possible (mid-right). For instance, task scheduling is subcontracted to some underlying RTOS, or tasks can be physically distributed.

Finally, task and even component scheduling can be entirely triggered by physical time, by using a distributed infrastructure which provides physically synchronized timers[1], this is usually referred to as "time-triggered architecture" [17].

Objective of this paper. As can be expected from the above discussion, mixed communication paradigms are in use throughout the design process, and are even combined both at early phases of the design and at deployment phase.

This was not so much an issue in the traditional design flow, in which most work was performed manually. In this traditional approach: the physics engineer provides models; the control engineer massages them for his own use and designs the control; then he forwards this as a document in textual/graphical format to the software engineer, who performs programming (in C or assembly language). This holds for each component. Then unit testing follows, and then integration and system testing [2]. Bugs discovered at this last stage are the nightmare of the systems designer! Where and how to find the cause? How to fix them? On the other hand, for this traditional design flow, each engineer has his own skills and underlying scientific background, but there is no need for an overall coherent mathematical foundation for the whole. So the design flow is simple. It uses different skills in a (nearly) independent way. This is why this is mainly the current practice.

However, due to the above indicated drawback, this design flow does not scale up. In very complex systems, many components would mutually interact in an intricate way. There are about 70 ECU's (Electronic Computing Units) in a modern BMW Series 7 car, each of these implements one or more functionalities. Moreover, some of them interact together, and the number of embedded functionalities rapidly increases. Therefore, there is a double need. First, specifications transferred between the different stages of the design must be as formal as possible (fully formal is the best). Second, the ancillary phases, such as programming, must be made automatic from higher level specifications [3].

[1] we prefer not to use the term *clock* for this, since the latter term will be used for a different purpose in the present paper.

[2] This is known as the traditional cycle consisting of {specification $\searrow$ coding $\searrow$ unit testing $\nearrow$ integration $\nearrow$ system testing}, with everything manual. It is called the V-shaped development cycle.

[3] Referring to Footnote 2, when some of the listed activities become automatic (e.g., coding being replaced by code generation), then the corresponding $\searrow\!\!\nearrow$ is replaced by a $\updownarrow$ (to refer to a "zero-time" activity), thus one moves from a V to a Y, and then further to a T, by relying on extensive virtual prototyping, an approach promoted by the Ptolemy tool [8].

This can only be achieved if we have a full understanding of how the different communication paradigms, attached to the different stages of the design flow, can be combined, and of how migration from a paradigm to the next one can be performed in a provably correct way. A study involving all the above mentioned paradigms is beyond the current state of the research. The purpose of this paper is to focus on the pair consisting of the {*synchronous, asynchronous*} paradigms.

But, before doing so, it is worth discussing in more depth the synchronous programming paradigm and its associated family of tools, as this paradigm is certainly not familiar to the High Performance Computing community. Although many visual or textual formalisms follow this paradigm, it is the contribution of the three "synchronous languages" Esterel, Lustre, and Signal [1] [7] [13] [18] [14] [6] [2], to have provided a firm basis for this concept.

3 Synchronous programming and synchronous languages

The three synchronous languages Esterel, Lustre, and Signal, are built on a common mathematical framework that combines synchrony (i.e., time progresses in lockstep with one or more clocks) with deterministic concurrency.

Fundamentals of synchrony. Requirements from the applications, as resulting from the discussion of Section 2, are the following:

- *Concurrency.* The languages must support functional concurrency, and they must rely on notations that express concurrency in a user-friendly manner. Therefore, depending on the targeted application area, the languages should offer as a notation: block diagrams (also called dataflow diagrams), or hierachical automata, or some imperative type of syntax, familiar to the targeted engineering communities.
- *Simplicity.* The languages must have the simplest formal model possible to make formal reasoning tractable. In particular, the semantics for the parallel composition of two processes must be the cleanest possible.
- *Synchrony.* The languages must support the simple and frequently-used implementation models in Fig. 3, where all mentioned actions are assumed to take finite memory and time.

Combining synchrony and concurrency while maintaining a simple mathematical model is not so straightforward. Here, we discuss the approach taken by the synchronous languages.

Synchrony divides time into discrete instants: a synchronous program progresses according to successive *atomic reactions,* in which the program communicates with its environment and performs computations, see Fig. 3. We write this for convenience using the "pseudo-mathematical" statement $P =_{\text{def}} R^{\omega}$, where R denotes the set of all possible reactions and the superscript ω indicates non-terminating iterations.

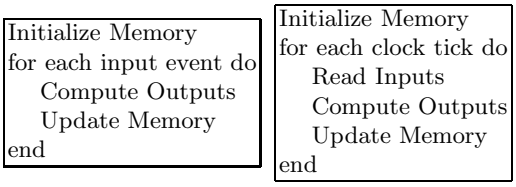

Figure3. Two common synchronous execution schemes: event driven (left) and sample driven (right). The bodies of the two loops are examples of *reactions.*

For example, in the block (or dataflow) diagrams of control engineering, the nth reaction of the whole system is the combination of the individual nth reactions for each constitutive component. For component i,

$$\begin{aligned} X_n^i &= f(X_{n-1}^i, U_n^i) \\ Y_n^i &= g(X_{n-1}^i, U_n^i) \end{aligned} \tag{1}$$

where U, X, Y are the (vector) input, state, and output, and combination means that some input or output of component i is connected to some input of component j, say

$$U_n^j(k) = U_n^i(l) \text{ or } Y_n^i(l), \tag{2}$$

where $Y_n^i(l)$ denotes the l-th coordinate of vector output of component i at instant n. Hence the whole reaction is simply the conjunction of the reactions (1) for each component, and the connections (2) between components.

Connecting two finite-state machines (FSM) in hardware is similar. Fig. 4a shows how a finite-state system is typically implemented in synchronous digital logic: a block of acyclic (and hence functional) logic computes outputs and the next state as a function of inputs and the current state. Fig. 4b shows the most natural way to run two such FSMs concurrently and have them communicate, i.e., by connecting some of the outputs of one FSM to the inputs of the other and vice versa.

Therefore, the following natural definition for parallel composition in synchronous languages was chosen, namely: $P_1 \parallel P_2 =_{\text{def}} (R_1 \wedge R_2)^\omega$, where $\wedge$ denotes conjunction. Note that this definition for parallel composition also fits several variants of the synchronous product of automata. Hence the model of synchrony can be summarized by the following two pseudo-equations:

$$P =_{\text{def}} R^\omega, \tag{3}$$

$$P_1 \parallel P_2 =_{\text{def}} (R_1 \wedge R_2)^\omega. \tag{4}$$

A flavour of the different styles of synchronous languages. Here is an example of a Lustre program, which describes a typical fragment of digital logic hardware. The program:

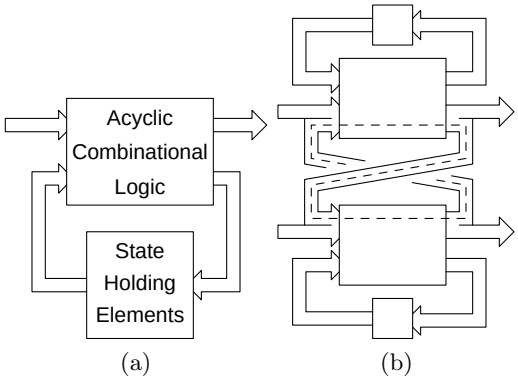

Figure4. (a) The usual structure of an FSM implemented in hardware. (b) Connecting two FSMs. The dashed line shows a path with instantaneous feedback that arises from connecting these two otherwise functional FSMs.

```
edge = false -> (c and not pre(c));
nat = 0 -> pre(nat) + 1;
edgecount = 0 -> if edge then pre(edgecount) + 1
                 else pre(edgecount);
```

defines edge to be true whenever the Boolean flow c has a rising edge, nat to be the step counter ($\mathsf{nat}_n = n$), and edgecount to count the number of rising edges in c. Its meaning can be expressed in the form of a finite difference equation, with obvious shorthand notations:

$$\begin{cases} e_0 = \mathit{false} \\ N_0 = 0 \end{cases}, \forall n > 0 : \begin{cases} e_n = c_n \text{ and not } c_{n-1} \\ N_n = N_{n-1} + 1 \\ ec_n = \begin{cases} \text{if } e_n = \mathit{true} \text{ then } ec_{n-1} + 1 \\ \text{else } \quad ec_{n-1} \end{cases} \end{cases}$$

This style of programming is amenable of graphical formalisms of block-diagram type. It is suited for *computation-dominated* programs. The Signal language is sort of a generalization of the Lustre language, suited to handle open systems, we discuss this point later on.

But reactive systems can also be *control-dominated*. To illustrate how Esterel can be used to describe control behavior, consider the program fragment in Fig. 5 describing the user interface of a portable CD player. It has input signals for play and stop and a lock signal that causes these signals to be ignored until an unlock signal is received, to prevent the player from accidentally starting while stuffed in a bag. Note how the first process ignores the Play signal when it is already playing, and how the suspend statement is used to ignore Stop and Play signals.

The nice thing about synchronous language is that, despite the very different styles of Esterel, Lustre, and Signal, they can be cleanly combined, since they share fully common mathematical semantics.

```
loop
  suspend
    await Play; emit Change
  when Locked;
  abort
    run CodeForPlay
  when Change
end
||
loop
  suspend
    await Stop; emit Change
  when Locked;
  abort
    run CodeForStop
  when Change
end
||
every Lock do
  abort
    sustain Locked
  when Unlock
end
```

emit S	Make signal S present immediately
pause	Stop this thread of control until the next reaction
p ; q	Run p then q
loop p end	Run p; restart when it terminates
await S	Pause until the next reaction in which S is present
p || q	Start p and q together; terminate when both have terminated
abort p when S	Run p up to, but not including, a reaction in which S is present
suspend p when S	Run p except when S is present
sustain S	Means loop emit S; pause end
run M	Expands to code for module M

Figure5. An Esterel program fragment describing the user interface of a portable CD player. Play and Stop inputs represent the usual pushbutton controls. The presence of the Lock input causes these commands to be ignored.

Besides the three so-called "synchronous languages", other formalisms or notations share the same type of mathematical semantics, without saying so explicitly. We only mention two major ones. The most widespread formalism is the discrete time part of the Simulink [4] graphical modeling tool for Matlab, it is a dataflow graphical formalism. David Harel's Statecharts [15][16] as for instance implemented in the Statemate tool by Ilogix [5], is a visual formalism to specify concurrent and hierarchical state machines. These formalisms are much more widely used than the previously described synchronous languages. However they do not fully exploit the underlying mathematical theory.

4 Desynchronization

As can be seen from Fig. 1, functionalities are naturally specified using the paradigm of synchrony. In contrast, by looking at the bottom part of the diagrams in the same figure, one can notice that, for larger systems, deployment uses

[4] http://www.mathworks.com/products/
[5] http://www.ilogix.com/frame_html.cfm

infrastructures that do not comply with the model of synchrony. This problem can be addressed in two different ways.

1. If the objective is to combine, in the considered system, functionalities that are only loosely coupled, then a direct integration without any special care taken to the nondeterminism of the distributed, asynchronous, infrastructure, will do the job. As an example, think of integrating an air bag system with an anti-skidding system in an automobile. In fact, integrating different functionalities in the overall system, is mostly performed this way in the current practice [11].
2. However, when different functionalities have to be combined, which involve a significant discrete part, and interact together in a tight way, then brute force deployment on a nondeterministic infrastructure can create unexpected combinations of discrete states, a source of risk. As an example to contrast with the previous one, think of combining an air bag system with an automatic door locking control (which decides upon locking/unlocking the doors depending on the driving condition).

For this second case, having a precise understanding of how to perform, in a provably correct way, asynchronous distributed deployment of synchronous systems, is a key issue. In this section, we summarize our theory on the interaction between the two {synchronous, asynchronous} paradigms [5].

4.1 The models used

In all the models discussed below, we assume some given underlying finite set V of variables—with no loss of generality, we will assume that each system possesses the same V as its set of variables. Interaction between systems occurs via common variables. The difference between these models lies in the way this interaction occurs, from strictly synchronous to asynchronous. We consider the following three different models:

- *Strictly synchronous*: Think of an intelligent sensor, it possesses a unique clock which triggers the reading of its input values, the processing it performs, and the delivery of its processed values to the bus. The same model can be used for human/machine interfaces, in which the internal clock triggers the scanning of the possible input events: only a subset of these are present at a given tick of the overall clock.
- *Synchronous*: The previous model becomes inadequate when open systems are considered. Think of a generic protection subsystem, it must perform reconfiguration actions on the reception of some alarm event—thus, "some alarm event" is the clock which triggers this protection subsystem, when being designed. But, clearly, this protection subsystem is for subsequent use in combination with some sensoring system which will generate the possible alarm events. Thus, if we wish to consider the protection system separately, we must regard it as an open system, which will be combined with some other, yet unspecified, subsystems. And these additional components may

very well be active when the considered open system is silent, cf. the example of the protection subsystem. Thus, the model of a global clock triggering the whole system becomes inadequate for open systems, and we must go for a view in which *several* clocks trigger different components or subsystems, which would in turn interact at some synchronization points. This is an extension of the strictly synchronous model, we call it synchronous. The Esterel and Lustre languages follow the strictly synchronous paradigm, whereas Signal also encompasses the synchronous one.

– *Asynchronous*: In the synchronous model, interacting components or subsystems share some clocks for their mutual synchronization, this requires some kind of broadcast synchronization protocol. Unfortunately, most distributed architectures are asynchronous and do not offer such a service. Instead, they would typically offer asynchronous communication services satisfying the following conditions: 1/ no data shall be lost, and 2/ the ordering of the successive values, for a given variable, shall be preserved (but the global interleaving of the different variables is not). This corresponds to a network of reliable, point to point channels, with otherwise no synchronization service being provided. This type of infrastructure is typically offered by RTOS or buses in embedded distributed architectures, we refer to it as an asynchronous infrastructure in the sequel.

We formalize these three models as follows.

Strictly synchronous. According to this model, a *state* x assigns an effective value to each variable $v \in V$. A *strictly synchronous behaviour* is a sequence $\sigma = x_1, x_2, \ldots$ of states. A *strictly synchronous process* is a set of strictly synchronous behaviours. A *strictly synchronous signal* is the sequence of values $\sigma_v = v(x_1), v(x_2), \ldots$, 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, where R is some transition relation. Therefore, strictly synchronous processes take the form

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

where superscript "$.^{\omega}$" denotes unbounded iterations[6]. Composition is defined as the intersection of the set of behaviours, it is performed by taking the conjunction of reactions:

$$P \parallel P' \;:=\; P \cap P' = (R \wedge R')^{\omega}\,. \tag{5}$$

This is the classical mathematical framework used in (discrete time) models in scientific engineering, where systems of difference equations and finite state

[6] Now, it is clear why we can assume that all processes possess identical sets of variables: just enlarge the actual set of variables with additional ones, by setting no constraint on the values taken by the states for these additional variables.

machines are usually considered. But it is also used in synchronous hardware modeling.

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 $\perp$ [3][4][5]. A $\perp$ value is to be interpreted as the considered variable being *absent* in the considered reaction And the process can use the absence of these variables as a viable information for its control. Besides this, things are as before: a *state* x assigns an informative or non-informative value to each state variable $v \in V$. A *synchronous behaviour* is a sequence of states: $\sigma = x_0, x_1, x_2, \ldots$. A *synchronous process* is a set of synchronous behaviours. A *synchronous signal* is the sequence of informative or non-informative values $\sigma_v = v(x_1), v(x_2), \ldots$, for $v \in V$ given. And composition is performed as in (5). Hence, strictly synchronous processes are just synchronous processes involving only informative (or "present") values.

A reaction is called *silent* if all variables are absent in the considered reaction. Now, if $P = P_1 \parallel P_2 \parallel \ldots \parallel P_K$ is a system composed of a set of components, each P_k has its own activation clock, consisting of the sequence of its non-silent reactions. Thus the activation clock of P_k is local to it, and activation clocks provide the adequate notion of local time reference for larger systems. For instance, if P_1 and P_2 do not interact at all (they share no variable), then there is no purpose that they should share some time reference. According to the synchronous model, *non interacting components simply possess independent, non synchronized, activation clocks.*

Thus, our synchronous model can mimic asynchrony. As soon as two processes can synchronize on some common clock, they can also exercise control on the basis of the absence of some variables at a given instant of this shared clock. Of course, sharing a clock needs broadcasting this clock among the different involved processes, this may require some protocol if the considered components are distributed.

Asynchronous. Reactions cannot be observed any more, no clock exists. Instead a *behaviour* is a tuple of *signals,* and each individual signal is a totally ordered sequence of (informative) values: $s_v = v(1), v(2), \ldots$ A process P 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:

$$P_1 \parallel_a P_2 := P_1 \cap P_2$$

Hence, in this model, a network of reliable and order-preserving, point-to-point channels is assumed (since each individual signal must be preserved by the medium), but no synchronization between the different channels is required. This models in particular the communications via asynchronous unbounded FIFOs.

4.2 The fundamental problems

Many embedded systems use the Globally Asynchronous Locally Synchronous (GALS) architecture, which consists of a network of synchronous processes, interconnected by asynchronous communications (as defined above). The central issue considered in this paper is:

> *what do we preserve when deploying a synchronous specification on a* GALS *architecture?*

The issue is best illustrated in Fig. 6. In this figure, we show a how desynchro-

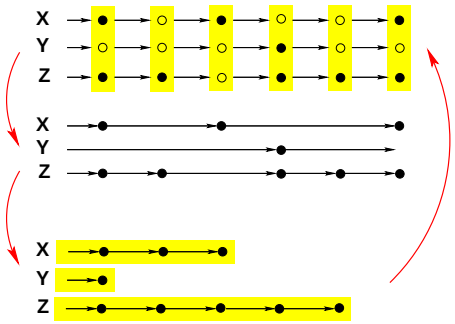

Figure6. Desynchronization / resynchronization. Unless desynchronization (shown by the downgoing arrows), resynchronization (shown by the upgoing arrows) is generally non determinate.

nization modifies a given run of a synchronous program. The synchronous run is shown on the top, it involves three variables, X, Y, Z. That this is a synchronous run is manifested by the presence of the successive rectangular patches, indicating the successive reactions. A black circle indicates that the considered variable is present in the considered reaction, and a white circle indicates that it is absent; for example, X is present in reactions $1, 3, 6$. Desynchronizing this run amounts to 1/ removing the global synchronization clock indicating the successive reactions, and 2/ erasing the absent occurrences, for each variable individually, since absence has no meaning when no more synchronization clock is available. The result is shown in the middle. And there is no difference between the mid and bottom drawings, since time is only logical, not metric. Of course, the downgoing arrows define a proper desynchronization map, we formalize it below. In contrast, desynchronization is clearly not revertible in general, since there are many different possible ways of inserting absent occurrences, for each variable.

Problem 1: What if a synchronous program receives its data from an asynchronous environment? Focus on a synchronous program within a GALS architecture, it receives its inputs as a tuple of (non synchronized) signals.

Since some variables can be absent in a given state, it can be the case that some signals 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 program the correct model for its input stimuli. In general this will drastically affect the semantics of the program. However, some particular synchronous programs are robust against this type of difficulty. How to formalize this?

Let P be such a program, we recall some notations for subsequent use. Symbol $\sigma = x_0, x_1, x_2, \ldots$ denotes a behaviour of P, i.e., a sequence of states compliant with the reactions of P. V is the (finite) set of state variables of P. Each state x is a valuation for all $v \in V$, the valuation for v at state x is written $v(x)$. Hence we can write equivalently

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

The valuation $v(x)$ is either an informative value belonging to some domain (e.g., boolean, integer), or it can be the possible special status *absent,* which is denoted by the special symbol $\perp$ in [3][4][5]. 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 [3][4][5] the first fundamental result that

> if P satisfies a special condition called *endochrony,* then
> $\forall \sigma^a \in P^a$ there exists a unique $\sigma \in P$ such that $\sigma \mapsto \sigma^a$ holds. (6)

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 [3][4][5] 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.

Examples/counterexamples. Referring to Fig. 3, the program shown on the left is *not* endochronous. The environment tells the program which input event is present in the considered reaction, thus the environment provides the structuration of the run into its successive reactions. An asynchronous environment would not provide this service.

In contrast, the program on the right is endochronous. In its simplest form, all inputs are present at each clock tick. In a more complex form, some inputs can be absent, but this the presence/absence, for each input, is *explicitly* indicated by some corresponding always present boolean input. In other words, clocks are encoded using always present booleans; reading the value of these booleans tells the program which input is present in the considered reaction. Thus no extra synchronization role is played by the environment, the synchronization is entirely carried by the program itself (hence the name).

Clearly, if, for the considered program, it is known that the absence of some variable X implies the absence of some other variable Y, then there is no need to read the boolean clock of Y when X is absent. Endochrony introduced in [3][4][5] generalizes this informal analysis. ◇

The important point about result (6) is that endochrony can be model-checked [7] on the reaction R defining the synchronous process P. Also,

$$\text{any } P \text{ can be given a } \textit{wrapper } W \text{ making } P \| W \text{ endochronous.} \qquad (7)$$

How can we use (6) 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 (6) 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 program interacting with its asynchronous environment. It remains to consider the problem of mapping a synchronous network of synchronous programs onto a GALS architecture.

Problem 2 : What if we deploy a synchronous network of synchronous programs onto a GALS architecture ? Consider the simple case of a network of two programs P and Q. Since our communication media behave like a set of FIFOs, one per signal sent from one program to the other, we already know what

[7] *Model checking* consists in exhaustively exploring the state space of a finite state model, for checking whether some given property is satisfied or not by this model. See [12].

the desynchronized behaviours of our deployed system will be, namely:

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

There is not need for inserting any particular explicit model for the communication medium, since by definition $\parallel_a$-communication preserves 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, 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 $P^a \parallel_a Q^a$.

Yet, we are not happy: it may be the case that there exists 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

$$(P \parallel Q)^a \subseteq (P^a \parallel_a Q^a). \tag{8}$$

However, we have shown in [3][4][5] the second fundamental result that

$$\begin{array}{l}\text{if } (P,Q) \text{ satisfies a special condition called } \textit{isochrony,}\\ \text{then equality in (8) indeed holds.}\end{array} \tag{9}$$

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 compositionality properties listed in [3][4][5]. Again, the condition of isochrony can be model-checked on the pair of reactions associated to the pair (P,Q), and

$$\begin{array}{l}\text{any pair } (P,Q) \text{ can be given } \textit{wrappers } (W_P, W_Q)\\ \text{making } (P\|W_P, Q\|W_Q) \text{ an isochronous pair.}\end{array} \tag{10}$$

Examples. A pair (P,Q) of programs having a single clocked communication (all shared variables possess the same clock), is isochronous. More generally, if the restriction of $P \parallel Q$, to the subset of shared variables, is endochronous, the the pair (P,Q) is isochronous: an isochronous pair does not need extra syncrhronization help from the environment, in order to communicate. ◇

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

$$R_P \wedge R_Q$$

of corresponding reactions of P and Q. 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 requiring 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 (6) and (9) 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 1. *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 programs. If the network satisfies conditions* (GALS$_1$) *and* (GALS$_2$)*, then the original synchronous semantics of each individual program 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 1.

5 Conclusion

There are important distributed computing systems which are neither massive nor high performance, systems of that kind are in fact numerous—they are estimated to constitute more than 80% of the computer systems. Still, their design can be extremely complex, and it raises several difficult problems of interest for computer scientists. These are mainly related to tracking the correctness of the implementation throughout the different design phases. Synchronous languages have emerged as an efficient vehicle for this, but the distributed implementation of synchronous programs raises some fundamental difficulties, which we have briefly reviewed.

Still, this issue is not closed, since not every distributed architecture in use in actual embedded systems complies with our model of "reliable" asynchrony [17]. In fact, the bus architecture used at Airbus does not satisfy our assumptions, and there are excellent reasons for this. Many additional studies are underway to address actual architectures in use in important safety critical systems [10][11].

Acknowledgement. The author is gratefully indebted to Luc Bougé for his help in selecting the focus and style of this paper, and to Joel Daniels for correcting a draft version of it.

References

1. A. Benveniste and G. Berry, The synchronous approach to reactive real-time systems. *Proceedings of the IEEE*, 79, 1270–1282, Sept. 1991.
2. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. To appear in *Proceedings of the IEEE,* special issue on Embedded Systems, Sastry and Sztipanovits Eds., 2002.
3. A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages : specification & distributed code generation. *Information and Computation*, 163, 125-171, 2000.
4. 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, *Lecture Notes in Computer Science*, vol. 1664, 162–177, Springer Verlag, 1999.
5. A. Benveniste. Some synchronization issuess when designing embedded systems. In Proc. of the first int. workshop on Embedded Software, EMSOFT'2001, T.A. Henzinger and C.M. Kirsch Eds., *Lecture Notes in Computer Science*, vol 2211, 32–49, Springer Verlag, 2001.
6. G. Berry, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, ch. The Foundations of Esterel. MIT Press, 2000.
7. F. Boussinot and R. de Simone, "The Esterel language," *Proceedings of the IEEE*, vol. 79, 1293–1304, Sept. 1991.
8. J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of computer Simulation*, special issue on Simulation Software Development, 1994.

9. L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. The theory of latency insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 20(9), Sept. 2001.
10. 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, Joseph Mathai Ed., *Lecture Notes in Computer Science,* vol. 1926, 68–81, Springer Verlag, Sept. 2000.
11. P. Caspi. Embedded control: from asynchrony to synchrony and back. In Proc. of the first int. workshop on Embedded Software, EMSOFT'2001, T.A. Henzinger and C.M. Kirsch Eds., *Lecture Notes in Computer Science,* vol 2211, 80–96, Springer Verlag, 2001.
12. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems,* 8(2), 244–263, April 1986.
13. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, 1305–1320, Sept. 1991.
14. N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.
15. D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, 231–274, June 1987.
16. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts.* McGraw-Hill, 1998.
17. H. Kopetz, *Real-time systems, design principles for distributed embedded applications, 3rd edition.* London: Kluwer academic publishers, 1997.
18. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, 1321–1336, Sept. 1991.
19. J. Rumbaugh, I. Jacobson, and G. Booch, *Tne Unified Modeling Language reference manual.* Object technologies series, Addison-Wesley, 1999.
20. J. Sztipanovits and G. Karsai. Embedded software: challenges and opportunities. In Proc. of the first int. workshop on Embedded Software, EMSOFT'2001, T.A. Henzinger and C.M. Kirsch Eds., *Lecture Notes in Computer Science,* vol 2211, 403–415, Springer Verlag, 2001.