A Formal Modeling Framework for Deploying Synchronous Designs on Distributed Architectures *

Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli

EECS Department, University of California at Berkeley, Berkeley, CA 94720
http://www-cad.eecs.berkeley.edu/HomePages/{lcarloni,alberto}

Abstract. Synchronous specifications are appealing in the design of large scale hardware and software systems because of their properties that facilitate verification *and* synthesis. When the target architecture is a *distributed system*, implementing a synchronous specification as a synchronous design may be inefficient in terms of both size (memory for software implementations or area for hardware implementations) and performance. A more elaborate implementation style where the basic synchronous paradigm is adapted to distributed architectures by introducing elements of asynchrony is, hence, highly desirable. This approach has to conjugate the desire of maintaining the theoretical properties of synchronous designs with the efficiency of implementations where the constraints imposed by synchrony are relaxed. Two interesting avenues have been recently pursued to achieve this goal:

- Latency-insensitive protocols [9,10] motivated by hardware implementations, where long paths between the design components may introduce delays that force the overall clock of the system to run too slow in order to maintain synchronous behavior. This approach introduces additional elements in the design to allow the implementation to maintain the throughput that could have been achieved with communication delays of the same order of the clock of the subsystems at the price of additional latency.
- Desynchronization [3,4,20] motivated by software implementations, where processes that compose the large scale system are locally implemented synchronously while their communication is implemented in an asynchronous style. This approach allows also to run each of the process at its own "speed".

By using the Lee and Sangiovanni-Vincentelli (LSV) tagged-signal model [19] as a common framework, we offer a comparative exposition of these approaches and we show their precise relationship. In doing so, we also provide some insight on the role of signal absence in synchronous, asynchronous, and globally-asynchronous locally-synchronous (GALS) design styles.

1 Introduction

The synchronous design paradigm is pervasive in electronic system engineering. It is used in discrete-time dynamical control systems, it is the basis of digital integrated circuit design, and it is the foundation of programming languages

* This research was supported in part by the NSF under the project ITR (CCR-0225610), the Gigascale System Research Center (GSRC), and by the European Commission under the projects COLUMBUS, IST-2002-38314, and ARTIST, IST-2001-34820.

and design environments used for software development for real-time embedded systems. In this paradigm, a complex system is represented as a collection of interacting modules whose state is updated collectively in one *zero-time* step. A synchronous specification is naturally simpler than specifying the same system as the interaction of components whose state is updated following an intricate set of time-based interdependency relations. However, for an increasing number of important applications, e.g., transportation systems, sensor networks and industrial control, the implementation architecture is distributed. In addition, the advent of deep-submicron (DSM) technologies for IC design, where hundreds of millions of transistors can be integrated on a single die, is making the synchronous paradigm very expensive to implement since the chip becomes a distributed system with interconnect delays that are up to an order of magnitude larger that the switching delays of the gates and that are *very difficult* to estimate [12].

In this scenario, we believe that new methodologies that combine specification simplicity with implementation constraints will take center place in the design stage. We are indeed confident that two main research themes, heterogeneity and desynchronization, will be very important to develop these methodologies. Heterogeneity in system design comes in at least two flavors: (1) a system naturally accommodates components of heterogeneous nature (analog/digital, synchronous/asynchronous, hardware/software), and (2) the same system is specified, optimized, and verified at various levels of abstraction in the path from specification to implementation using different models of computation. The desynchronization problem can be informally described as the task of deploying a synchronous design on a distributed architecture in a correct-by-construction (and mostly automatic) fashion. The relevance of this problem follows naturally from the desire of leveraging the well-known tools and practices of synchronous design for the specification and the optimization of a system, while targeting efficient final implementations that are distributed in nature.

We present a modeling framework that addresses both facets of heterogeneity while focusing on the synchronization aspects of system design. Our framework encompasses different design styles from the "strong assumptions" of synchronous design and asynchronous design, to more "relaxed and realistic" models for distributed design, like GALS. We argue for the importance of the notions of absence to distinguish (and relate) these systems, and we illustrate their interplay in modeling the desynchronization problem. Finally, we revisit previous work on distributed embedded code generation (desynchronization) and latency-insensitive design and we elaborate on possible options to combine their results.

The desynchronization problem was formally defined in [2,3,20] and recently has been the object of investigation of several projects [4,13,15,16,21,22]. Latency-insensitive protocols were proposed in [9] and, then, applied to synchronous hardware design in [8]. A complete presentation of the theory of latency-insensitive design is given in [10].

2 The Tagged-Signal Model

In this section, we summarize the main concepts of the Lee and Sangiovanni-Vincentelli's (LSV) tagged-signal model [19], the basis of our formal framework.

Given a set of *values* $\mathcal{D}$ and a set of *tags* $\mathcal{T}$, an *event* is a member of $\mathcal{D}\times\mathcal{T}$. A *signal* s is a set of events. The set of all M-tuples of signals is denoted $\mathcal{S}^M$ and a *process* P is a subset of $\mathcal{S}^M$. A particular M-tuple $b = (s_1, \ldots, s_M) \in \mathcal{S}^M$ satisfies the process if $b \in P$. An M-tuple b that satisfies a process is called a *behavior* of the process. Thus, a process is a set of possible behaviors. A *tagged system* is a composition of processes $\{P_1, \ldots, P_I\}$, that is a new process P that is defined as the intersection of their behaviors $P = \bigcap_{i=1}^{I} P_i$. To distinguish signals, we assume an underlying set $\mathcal{V}$ of variables with domain $\mathcal{D}$. We denote a tagged system as a triple $P = (V, \mathcal{T}, \mathcal{B})$, where $V \subset \mathcal{V}$ is a finite set of variables, $\mathcal{T}$ is a tag set, and $\mathcal{B}$ a set of behaviors with domain V. As we anticipated, *composition* of two systems P_1 and P_2 is given by the intersection of their behaviors:

$$\begin{aligned} &P_1 \cap P_2 =_{\text{def}} (V_1 \cup V_2, \mathcal{T}_1 \cup \mathcal{T}_2, \mathcal{B}_1 \cap \mathcal{B}_2), \text{ where} \\ &\mathcal{B}_1 \cap \mathcal{B}_2 =_{\text{def}} \left\{ b \,\middle|\, b_{|V_i} \in \mathcal{B}_i, i = 1, 2 \right\}, \end{aligned}$$

and $b_{|W}$ denotes the restriction of b to a subset W of variables. In the sequel, we denote with $\mathcal{T}(s)$ the tag set of a signal s (and, similarly, for a behavior and a process).

In some models of computation the set $\mathcal{D}$ includes a special value $\perp$, which indicates the absence of a value. For any tag $t \in \mathcal{T}$, we call $(t, \perp)$ the *absent-value event*, or simply, $\perp$ event. We say that a signal s is *present* at a given tag t when $(\exists e = (t, d) \in s \mid d \neq \perp)$; otherwise, we say that s is *absent* at t (or, that s has an *event absence* at t).

Example 1. The following diagram represents the unique behavior of a system that has two signals with names u, v. At any given tag, signal u is present with unit value if and only if signal v is present and carries a positive integer value.

	tag :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
P :	u :	4	-2	5	$\perp$	-1	3	4	2	...
	v :	1	$\perp$	1	$\perp$	$\perp$	1	1	1	...

Assumption 1 *For any tag $t \in \mathcal{T}$, each signal s in the system has at most one event, i.e.:*

$$\forall b \in \mathcal{B}, \forall s \in b,\ \neg\Big[\exists e_1 \in s, \exists e_2 \in s \mid tag(e_1) = tag(e_2)\Big]$$

Ordering among Signal Tags. Assumption 1 logically implies a total order $<$ among the tags of *a signal*. Then, the total order over the tag set $\mathcal{T}(s)$ of signal s induces a total order among its events. Therefore, a signal can be seen as a sequence of events. In the sequel, we use the notation t_i to denote the i-th tag of a signal and, naturally, we rely on the fact that $t_i < t_j \Leftrightarrow i < j$. Further, we can use tags to identify an event of a signal (somewhat like the indexes of an array) as well as its values. Given a signal s and a tag t, we write $e = eve(s, t)$ to denote the event of s whose tag is t and we write $d = val(s, t)$ to denote the value of $eve(s, t)$.

The set of all sequences of elements in $\mathcal{D} \cup \{\bot\}$ is denoted by Σ. Function $\sigma : \mathcal{S}^1 \times \mathcal{T}^2 \rightarrow \Sigma$ takes a signal $s = \{(d_0, t_0), (d_1, t_1), ..\}$ and an ordered tag pair $(t_i, t_j),\ i \leq j$, and returns a sequence $\sigma_{[t_i,t_j]} \in \Sigma$ s.t. $\sigma_{[t_i,t_j]}(s) = d_i, d_{i+1}, \ldots, d_j$. The sequence of values of a signal is denoted $\sigma(s)$. The empty sequence is denoted as ϵ. To manipulate sequences of values we define the filtering operator $\mathcal{F}_\bot : \Sigma \rightarrow \mathcal{D}$ that returns a sequence $\sigma' = \mathcal{F}_\bot[\sigma]$ s.t.

$$\sigma'_i = \begin{cases} \sigma_{[t_i,t_i]}(s) & \text{if } \sigma_{[t_i,t_i]}(s) \in \mathcal{D} \\ \epsilon & \text{if } \sigma_{[t_i,t_i]}(s) = \bot \end{cases}$$

Ordering among Process Tags. In general, the tag set $\mathcal{T}$ of a process is not ordered, let alone totally ordered. When tags are used to express causality relations among signals, it is common to assume that $\mathcal{T}$ is partially ordered. In this case, $\leq$ is used to denote the partial order on $\mathcal{T}$ by writing $t < t'$ when $t \leq t'$ and $t \neq t'$. Finally, a tag system is *timed* if $\mathcal{T}$ is a totally ordered set, i.e. for each pair of distinct tags t, t' either $t < t'$ or $t' < t$. Often tags are used as a mechanism to express time. This may be useful, for instance, to move across the various representations of a design at different levels of abstraction from initial specification, where *logical time* is central, to final implementation, where each event occurs at a given instant of the *physical*, or *real, time*. However, tags are essentially a tool to express constraints, like coordination constraints, among events of different signals (and, transitively, among signals and among processes).

3 Models of Computation

We use models of computation to specify the mathematical behavior of the systems under design [14]. The models of computation addressed in this paper fall

	tag :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
	w :	1	0	1	0	1	0	1	0	...
P :	y :	0	2	2	6	6	10	10	14	...
	z :	0	0	4	0	8	4	12	8	...
	w :	1	0	1	0	1	0	1	0	...
Q :	x :	1	3	5	7	9	11	13	15	...
	y :	0	2	2	6	6	10	10	14	...
	w :	1	0	1	0	1	0	1	0	...
R :	x :	1	3	5	7	9	11	13	15	...
	z :	0	0	4	0	8	4	12	8	...

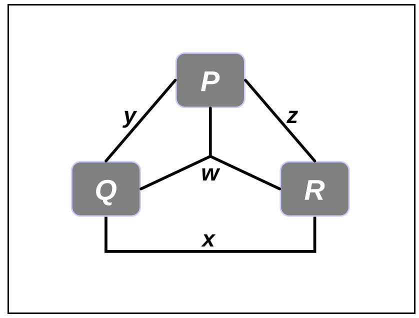

Fig. 1. The synchronous system of Example 2 and its behavior.

under the category of synchronous, asynchronous, and *in between* to indicate models that are neither.

3.1 Synchronous Systems

Two events e_1, e_2 are *synchronous* ($e_1 \approx e_2$) when they have the same tag, i.e. $e_1 \approx e_2 \Leftrightarrow tag(e_1) = tag(e_2)$. Two signals s_1, s_2 are synchronous ($s_1 \approx s_2$) when for each event of s_1 there is a synchronous event in s_2 and vice versa, i.e.:

$$s_1 \approx s_2 \;\Leftrightarrow\; \Big(\forall e_i \in s_1, \exists e_j \in s_2, \;\mid\; e_i \approx e_j\Big) \wedge \Big(\forall e_k \in s_2, \exists e_l \in s_1, \;\mid\; e_k \approx e_l\Big)$$

Therefore, synchronous signals share the tag set. The definitions of two synchronous behaviors b_1, b_2 and two synchronous processes $P_1 = (V_1, \mathcal{T}_1, \mathcal{B}_1)$, $P_2 = (V_2, \mathcal{T}_2, \mathcal{B}_2)$ naturally follow:

$$b_1 \approx b_2 \;\Leftrightarrow\; \forall s_i \in b_1, \forall s_j \in b_2, s_i \neq s_j,\; (s_i \approx s_j)$$
$$P_1 \approx P_2 \;\Leftrightarrow\; \forall b_i \in \mathcal{B}_1, \forall b_j \in \mathcal{B}_2,\; (b_i \approx b_j)$$

A stand-alone behavior b is synchronous when $b \approx b$. A stand-alone process P is synchronous when $P \approx P$. Observe that in a behavior of a synchronous system, every signal is synchronous with every other signal and, equivalently, for each tag a signal has *exactly one* corresponding event: $\forall b \in \mathcal{B}, \forall s \in b, \forall t \in \mathcal{T},\; (\exists! e \in s \mid tag(e) = t)$.

Example 2. The diagram of Figure 1 represents the unique behavior of a synchronous system that is the result of the composition of three processes P, Q, and R. Signal w, a binary, is shared by all processes, while the remaining signals, integers x, y, and z, are shared in pairwise manner. In Figure 1, the signals are purposely represented by simple lines and not arrows. In fact, by observing only the event sequences we can not say which input/output relations exist among the system processes. However, in the sequel, we focus our attention on functional systems [19] and we use this example assuming that signal w is produced by process P, signals x, y by process Q, and signal z by process R.

3.2 Synchronous Languages.

Synchronous programming languages like ESTEREL, LUSTRE, and SIGNAL represent powerful tools for the specification of complex real-time embedded systems because they allow to combine the simplicity of the synchronous assumption with the power of concurrency in functional specification [6,7,18,17]. They are synchronous systems with particular properties and for this reason, they are often considered a model of computation in addition to the generic synchronous model. The *synchronous programming model* can be expressed by the following "pseudo-mathematical" statements [3,5]:

$$P \equiv R^\omega$$
$$P_1 || P_2 \equiv (R_1 \wedge R_2)^\omega$$

where P, P_1, P_2 denote synchronous programs, R, R_1, R_2 denote the sets of all the possible reactions of the corresponding programs, and the superscript ω indicates non-terminating iterations. The first expression interprets the essence of the synchronous assumption: a synchronous program P evolves according to an infinite sequence of successive atomic reactions. At each reaction, the program variables may or may not present a value. The second expression defines the parallel composition of two components as the conjunction of the reactions for each component. This implies that communication among components is performed via instantaneous broadcast. To cast the synchronous programming model into the LSV formalism, we naturally associate signals to variables and use tags to index the program reactions. An important feature offered by the synchronous programming model is the ability of taking decisions based on the absence of a value for a variable at a given reaction, i.e., *in synchronous systems absence can be sensed*. This is perfectly in line with the definition of the absent-value event since processes react to events and hence can also react to the particular absent-value event. The absent-value event plays an important role in synchronous models of computation. In fact, the essence of the model is that all computation processes awake simultaneously when any of them posts an event for communication. Some of the signals that connect the processes may be not present. The synchronous model requires that these signals be read with the absent-value event posted. If indeed the information on the presence of an absent-value event does not cause a process to react to it, then reading this event is an unnecessary complication. We shall see later that recognizing this situation and eliminating the associated steps are key in deriving a more effective deployment that, while formally giving up the synchronous model, maintains behavior equivalence with the original synchronous specifications.

The notion of *clock of a variable* is introduced as a *Boolean meta-variable* tracking the absence/presence of a value for the corresponding variable [1]. Variables that are always present simultaneously are said to have the same clock, so that clocks can be seen as equivalence classes of simultaneously-present variables. In the sequel, we focus our attention on SIGNAL, which is a declarative language [6]. Besides parallel composition, SIGNAL's main operators are the followings:

- statement `c := a` *`op`* `b`, where *op* denotes a generic logic or arithmetic operator, defines not only that the values of `c` are function of those of `a` and `b`, but also that the three variables have the same clock;
- statement `c := a$`*`k`*, where `k` is a positive integer constant, specifies both that `c` and `a` have the same clock and that at the *n*-th reaction when the two signals are present, the value of `c` is equal to the value held by `a` at the (*n*-*k*)-th reaction;
- statement `c := a default b` specifies that variable `c` is present at every reaction where either `a` or `b` is present while taking the value of `b` only if `a` is not present (*oversampling*);
- statement `c := a when b` specifies that variable `c` is present (taking the value of `a`) only when both `a` is present and the Boolean condition expressed by variable `b` is true (*undersampling*).

While the first two statements are *single-clock*, the last two are *multi-clock*. Additional operators are available to directly relate the variable clocks: for instance, statement `c ^= a` constraints variables `c` and `a` to have the same clock, without relating the values that they assume. The SIGNAL compiler uses *clock calculus* to statically analyze every program statement, identify the structure of the clock of each variable, and schedule the overall computation. The compiler rejects the program when it detects that the collection of its statements as a whole contains clock constraint violations.

Example 3. Figure 2 reports the code of a SIGNAL program that is structured as a main process with three sub-processes $P, Q,$ and R. These processes communicate via signals w, x, y, z that are constrained to be synchronized (first statement of the main process). Hence, using SIGNAL jargon, these signals belong

[1] Notice that despite its name the clock of a variable is not necessarily a periodic signal.

```
process MAIN (
  ? boolean tag;
  ! boolean w, x, y, z;) (|
  | x ^= y ^= z ^= w ^= tag
  | w := P(tag, y, z)
  | (x,y) := Q(tag, w)
  | z := R(tag, w, x)
  |);
```

```
process P (
  ? boolean tag; integer y, z;) (|
  ! integer w;) (
  | i ^= tag
  | i := (i$1 init (-1)) + 1
  | iW := true when (i < 1)
  | w := iW default (y$1 > z$1)
  |) where integer i, iW; end;
```

```
process Q (
  ? boolean tag; integer w;
  ! integer x, y;) (|
  | i ^= tag
  | i := (i$1 init (-1)) + 1
  | iY := 0 when (i < 1)
  | y := iY default (
     if w$1 then (x$1+1) else (x$1-1))
  | x := (x + 2) $1 init 1
  |) where integer i, iY; end;
```

```
process R (
  ? boolean tag; integer w, x;
  ! integer z;) (|
  | i ^= tag
  | i := (i$1 init (-1)) +1
  | iZ := 0 when (i < 2)
  | z := iZ default (
     if w$1 then x$2 -3 else x$2 +3)
  |) where integer i, iZ;
  end;
```

Fig. 2. SIGNAL program with a deterministic behavior as in Example 2.

to the same *clock equivalence class* [6], which is also the class of signal tag. Signal tag is an external input whose values evolve as an infinite alternating sequence of 0s and 1s. Under this assumption, a run of program MAIN returns deterministically a tuple of sequences of values for variables w, x, y, z that coincide with the behavior of the synchronous system of Example 2. By analyzing the program we derive the functional relationships between its signals: for instance, we see that y and z are input signals for process P, which produces output signal w. Also, we learn causality dependencies among signals like, for instance, that every event of signal w, besides the first, depends on the events of y and z occurred at the previous reaction. Similarly, while the first two events of z carry values equal to 0, each subsequent event depends on the event occurred on w at the previous reaction as well as on the event occurred on x two reactions earlier. Hence, events of w depend on events of z and vice versa. In fact, cyclic causality dependencies across signals of a synchronous program are quite common and may be problematic only in the presence of a *combinational cycle*, i.e. when two events with the same tag depend on each other. The discussion of methods to handle this issue goes beyond the scope of this paper (see [5]).

3.3 Asynchronous Systems

The definition of asynchrony as used in the literature is vague: some use the term to indicate any systems that is *not* synchronous, others are more restrictive. According to [19], two events e_1, e_2 are *asynchronous* ($e_1 \simeq e_2$) if they have different tags, i.e. $e_1 \simeq e_2 \Leftrightarrow tag(e_1) \neq tag(e_2)$. Two signals s_1, s_2 are asynchronous ($s_1 \simeq s_2$) when:

$$s_1 \simeq s_2 \;\Leftrightarrow\; \Big(\forall e_i \in s_1 \;\; \nexists e_j \in s_2 \;\mid\; e_i \approx e_j\Big)$$

Asynchronous signals have disjoint tag sets. The definitions of asynchronous behaviors b_1, b_2 and asynchronous processes $P_1 = (V_1, \mathcal{T}_1, \mathcal{B}_1), P_2 = (V_2, \mathcal{T}_2, \mathcal{B}_2)$ follow:

$$b_1 \simeq b_2 \;\Leftrightarrow\; \forall s_i \in b_1, \forall s_j \in b_2, \; s_i \simeq s_j$$
$$P_1 \simeq P_2 \;\Leftrightarrow\; \forall b_i \in \mathcal{B}_1, \forall b_j \in \mathcal{B}_2, \; b_i \simeq b_j$$

A stand-alone behavior b is asynchronous when $b \simeq b$. A stand-alone process P is asynchronous when $P \simeq P$. In a behavior of an asynchronous system, every signal is asynchronous with every other signal and, equivalently, for each tag there is one and only one event across all signals: $\forall b = (s_1, \ldots, s_M) \in \mathcal{B}, \forall t \in \mathcal{T}, \; (\exists! e \in \bigcup_i s_i \mid tag(e) = t)$.

Example 4. The following diagram represents the unique behavior of the asynchronous system $S_a = P_a \cap Q_a \cap R_a$. Processes P_a, Q_a, and R_a communicate by sharing signals (as it is the case for synchronous systems), but signals do not share tags.

	tag :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	…
	w_a :	1				0				1				0				1				…
P_a :	y_a :			0				2				2				6				6		…
	z_a :				0				0				4				0				8	…
	w_a :	1				0				1				0				1				…
Q_a :	x_a :		1				3				5				7				9			…
	y_a :			0				2				2				6				6		…
	w_a :	1				0				1				0				1				…
R_a :	x_a :		1				3				5				7				9			…
	z_a :				0				0				4				0				8	…

3.4 Between Synchronous and Asynchronous: Globally-Asynchronous Locally-Synchronous Systems

Formally, the set of asynchronous systems is *not* the complement of the set of synchronous systems. In fact, there is a set of systems that sits *in between* these two sets and whose elements are useful to model heterogeneous systems and distributed architectures. An element of this *in-between set* is a process with a behavior that has both at least a pair of synchronous events (hence, it is not asynchronous) and at least a tag for which a signal does not present a corresponding event while another does (hence, it is not synchronous). A relevant subset of this set is the class of Globally-Asynchronous Locally-Synchronous (GALS) Systems.

GALS systems are of particular interest because they represent a compromise that allows designers to leverage the traditional practices and tools of synchronous design for implementations of synchronous processes on distributed architectures. In a GALS system, computation occurs in synchronous clusters exchanging data asynchronously via a set of communication media. Each cluster runs with its own clock that controls also the sampling of new values for its input signals. At each sampling period, some of these new values may or may not be present, depending on the transferring latencies in the asynchronous communication media. Since we want to focus on the communication mechanisms at the interface between synchronous and asynchronous, our LSV definition for GALS systems assumes, without lack of generality, that all asynchronous communications can be modeled as occurring within a single media process. A GALS system $S_g = \bigcap_{P_i \in \mathcal{P}} P_i \cap E$ is the composition of a collection $\mathcal{P}$ of *computation processes* and one *communication, or media, process* $E = (V_e, \mathcal{T}_e, \mathcal{B}_e)$ s.t.:

$$\forall P_i, P_j \in \mathcal{P},\ \Big((i = j \Rightarrow P_i \approx P_j) \wedge (i \neq j \Rightarrow P_i \simeq P_j)\Big),\ \text{and}$$
$$\forall P_i = (V_i, \mathcal{T}_i, \mathcal{B}_i), \forall b \in \mathcal{B}_e,\ \Big(b_{|V_i} \in \mathcal{B}_i\Big)$$

Each computation process is a stand-alone synchronous process because it runs with its own logical clock whose occurrences are represented by tags. In the general case, we assume that no relation exists between the clocks of distinct computation processes leaving total freedom in the implementation process. This is captured by saying that intersection of their tag sets is empty (i.e., they are pairwise asynchronous processes). Instead, a media process is not synchronous (because it models the communication latency and the sharing of communication resources among processes that are pairwise asynchronous) nor asynchronous (because each subset of its signals that interfaces a specific computation process is a synchronous sub-process). Hence, from a LSV perspective, the name globally-asynchronous locally-synchronous is justified when considering the system from the viewpoint of any of its computation processes.

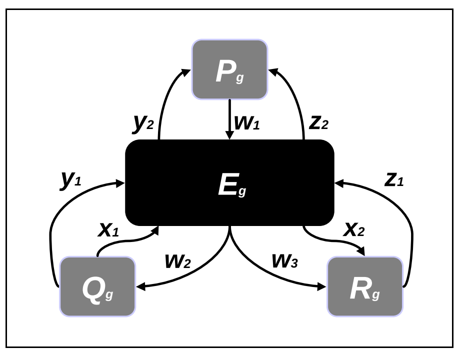

Fig. 3. GALS system for Example 5.

Example 5. The diagram below reports the unique behavior of the GALS system S_g of Figure 3, which is the result of the composition of four processes P_g, Q_g, R_g, and E_g. Process P_g is synchronous because all its signals are synchronous. The same is true for processes Q_g and R_g separately. However, the composition of processes P_g, Q_g, and R_g is not a synchronous process (no tag is shared across signals of different processes). Observe that P_g has a period twice as fast as those of Q_g and R_g. Processes P_g, Q_g, and R_g communicate only via the media process E_g. Process E_g, acting as the communication environment, has subsets of its signals synchronized with signals of the other processes, but, as a stand-alone process, it is not synchronous. The signals of E_g can be partitioned in equivalence classes, whose members carry the same sequence of values at different tags (e.g., signal x_2 is a "delayed" version of signal x_1). We call this relation semantic equivalence (see Section 4.1).

	tag :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	...
	w_1 :	1		$\perp$		$\perp$		0		$\perp$		1		$\perp$		$\perp$		0		$\perp$		1		$\perp$	...
P_g :	y_2 :	$\perp$		0		$\perp$		2		$\perp$		$\perp$		$\perp$		2		$\perp$		6		$\perp$		6	...
	z_2 :	$\perp$		$\perp$		0		$\perp$		0		$\perp$		4		$\perp$		0		$\perp$		$\perp$		$\perp$	...
	w_2 :		1				$\perp$				0				1				0				1		...
Q_g :	x_1 :		1				3				$\perp$				5				7				9		...
	y_1 :		0				2				$\perp$				2				6				6		...
	w_3 :				1				0				1				$\perp$				0				...
R_g :	x_2 :				1				3				$\perp$				5				7				...
	z_1 :				0				0				4				0				$\perp$				...
	w_1 :	1		$\perp$		$\perp$		0		$\perp$		1		$\perp$		$\perp$		0		$\perp$		1		$\perp$	...
	w_2 :		1				$\perp$				0				1				0				1		...
	w_3 :				1				0				1				$\perp$				0				...
	x_1 :		1				3				$\perp$				5				7				9		...
E_g :	x_2 :				1				3				$\perp$				5				7				...
	y_1 :		0				2				$\perp$				2				6				6		...
	y_2 :	$\perp$		0		$\perp$		2		$\perp$		$\perp$		$\perp$		2		$\perp$		6		$\perp$		6	...
	z_1 :				0				0				4				0				$\perp$				...
	z_2 :	$\perp$		$\perp$		0		$\perp$		0		$\perp$		4		$\perp$		0		$\perp$		$\perp$		$\perp$	...

Discussion on Communication Media. Our proposal for modeling a distributed system with the LSV model is to use more than one signal to capture each communication thread between two processes. For instance, if process P_g sends data to process Q_g, we must be able to distinguish between the sending event and the receiving event. To do so, we need at least two signals, e.g. w_1 and w_2. Each new event of w_1 is created by P_g, whose overall activity of reading input events and computing output events proceeds according to its tag set $\mathcal{T}(P_g)$. Then, a new event of w_1 causes at least a corresponding event of w_2 within the media process (more events could be necessary to model arbitrary latencies or the sharing of communication resources). Finally, event w_2 is consumed by Q_g, whose activity is controlled by tag set $\mathcal{T}(Q_g)$ that has empty intersection with the tag set of every other synchronous processes, including $\mathcal{T}(P_g)$. In synchronous systems, signal decoupling is not necessary thanks to the power of the synchronous abstraction: all processes create and sample events at the same tags and a unique signal w is sufficient to express the *instantaneous* communication [2], between process P and process Q (see Example 2). Strictly asynchronous systems rely on an abstraction that is equally powerful: there is no notion of global (i.e., system) or local (i.e., process) tag set and two processes communicate by sharing signals that are produced and sampled independently from the rest of the communication and computation activities in the system. The sharing of a signal in asynchronous systems represents the presence of an *ad-hoc* handshaking communication protocol, which is dedicated to the particular communication of, say, w_a from P_a to Q_a: a new event for w_a is created by P_a only when Q_a is ready to the sample it (see Example 4).

GALS and Absence. In the GALS system $S_g = (V_g, \mathcal{T}_g, \mathcal{B}_g)$ of Example 5, at any given tag $t \in \mathcal{T}_g$, for any signal $s \in V_g$, one of three things can happen:

1. $t \notin \mathcal{T}(s)$ *(event absence)*
2. $t \in \mathcal{T}(s) \;\wedge\; \exists e = (t, d) \in s \mid d = \perp$ *(value absence)*
3. $t \in \mathcal{T}(s) \;\wedge\; \exists e = (t, d) \in s \mid d \neq \perp$ *(presence)*

From a global viewpoint, a GALS system is a system with multiple tag sets (a multi-clock system), each representing a dimension that is familiar to a synchronous process and extraneous to all the remaining synchronous processes. For instance, the signals of process P_g do not have events at tag t_1 and the signals of processes Q_g and R_g do not have events at tag t_0. However, process P_g does not "expect" an event at tag t_1 nor at tags $t_3, t_5, \ldots$ because these are tags that do not belong to the tag set of P_g: they represent instants of a time dimension that is completely extraneous to P_g. Different meaning has the absence that P_g detects on signal y_2 at t_4, which is a tag belonging to $\mathcal{T}(P_g)$. In fact, at tag t_4, P_g is looking for a significant event, but it ends up sampling the absent-value event (the awaited event will arrive only at tag t_6, after being created by process Q_g) because of the latency introduced by the communication medium. This case is typical of a GALS system, because, in a purely synchronous model, the absent value event is always an "expected" event. In this case, the computation can be

[2] Instantaneous has to be interpreted in the sense of a process that is not "seen" by the computation part of the system. Communication and computation in synchronous systems never overlap.

affected in a substantial way since P_g can compute on the absence value event and produce an output that is different from the one originally expected.

Deploying automatically a synchronous design on a distributed architecture entails the development of techniques for making each process robust with respect to absence. In other words, we ask under which conditions we can guarantee that sensing the absent value event when a different value was expected does not produce incorrect behavior or that *not* sensing an absent value event when one is expected, does not change the behavior of the system. Section 4 discusses methods to achieve this result.

Remark. Consider again the case of asynchronous design (see Example 4). By definition, the tag sets of any two asynchronous signals x_a, y_a are disjoint. Thus, for each tag in $\mathcal{T}(x_a)$ there is no corresponding event in signal y_a and vice versa. If we consider an asynchronous system with several signals, we have that for each event that is present in one of its signal, there are absences in all the remaining signals. This phenomenon is so inherent to the notion of asynchronous system that here the notion of event absence looses its meaning: when events are always systematically absent, there is no point in looking for their presence! In fact, in asynchronous systems no common references exist across processes and processes cannot (and do not attempt) to detect event absence: inter-process communication occurs according to handshake protocols that don't leave space for this kind of uncertainty.

4 Deploying Synchronous Design on Distributed Architectures

In this section, we revisit previous research that targets the implementation of a synchronous design on a distributed architecture both in software and in hardware. We introduce the definition of semantic equivalence, which provides a formal ground to establish when the behavior of a distributed implementation conforms to the one of the synchronous specification. Then, we summarize the theory of latency-insensitive design and we present the main results on the desynchronization of synchronous programs for distributed embedded code generation. Finally, we compare these two approaches and we sketch two possible research avenues to combine them.

4.1 Semantic Equivalence

With the notion of semantic equivalence between processes we capture the fact that their operations produce exactly the same result from the viewpoint of an observer watching the sequences of values of their signals.

Two signals are *semantic equivalent* if they have the same sequence of values after discarding the $\perp$ events. This is written: $s \equiv s' \quad \Leftrightarrow \quad \mathcal{F}_{\perp}[\sigma(s)] =$

$\mathcal{F}_\perp[\sigma(s')]$. Two behaviors $b = (s_1, \ldots, s_M), b' = (s'_1, \ldots, s'_M)$ are semantic equivalent $b \equiv b'$ when there exists a bijective map $\psi : b \mapsto b'$ s.t. $\forall i, (s_i \equiv \psi(s'_i))$. Finally, for two processes $P = (V, \mathcal{T}, \mathcal{B}), P' = (V, \mathcal{T}', \mathcal{B}')$ we have: $P \equiv P' \Leftrightarrow (\forall b \in \mathcal{B}, \exists b' \in \mathcal{B}' \mid b \equiv b')$.

Similarly to *flow equivalence* [16], semantic equivalence indicates that two behaviors have exactly the same sequence of present events, which, however, may be interleaved by a different number of $\perp$ events. Hence, we can use it to model implicitly the communication latency between processes: e.g., the communication of events from P to Q occurs via a media process E and involves several signals $u_p, u_{e_1}, u_{e_2}, \ldots, , u_q$ that belong all to the same class of semantic equivalence. Observe that semantic equivalence doesn't say anything about tags: processes P, P' can be semantic equivalent even if $\mathcal{T}(P) \cap \mathcal{T}(P') = \emptyset$. Finally, note that it is a compositional property, in the sense that if two pairs of processes are semantic equivalent, their pairwise intersections give semantic equivalence processes, i.e. $(P \equiv P' \wedge Q \equiv Q') \Rightarrow (P \cap Q \equiv P' \cap Q')$.

Example 6. Reconsidering the systems of Examples 2 and 4, we have that $\psi(w) = w_a$, $\psi(x) = x_a$, $\psi(y) = y_a$, $\psi(z) = z_a$. and, consequently: $P \equiv P_a, Q \equiv Q_a, R \equiv R_a$ and, finally, $P \cap Q \cap R \equiv P_a \cap Q_a \cap R_a$. Now, consider the GALS system of Example 5. First, observe how semantic equivalence models the communication among the computation processes P_g, Q_g and R_g via the media process E_g. For instance, $\{w_1, w_2, w_3\}$ is a semantic equivalence class representing the communication from P_g to both Q_g and R_g. The other equivalence classes are $\{x_1, x_2\}, \{y_1, y_2\}$, and $\{z_1, z_2\}$. Let $V^\star = \{w_1, x_1, y_1, z_1\}$ be the set of representative variables for each equivalence class and for all behaviors $b \in P_g \cap Q_g \cap R_g$ let $b^\star = b_{|V^\star}$. Then, we have that $\psi'(w_1) = w$, $\psi'(x_1) = x$, $\psi'(y_1) = y$, $\psi'(z_1) = z$, and, finally: $(P_g \cap Q_g \cap R_g)_{|V^\star} \equiv P \cap Q \cap R$.

4.2 Latency-Insensitive Systems.

Latency-insensitive protocols [9,10] were originally proposed to address the interconnect delay problem in synchronous hardware design. The latency-insensitive design methodology takes a *strict synchronous* system specification and automatically derives a *latency-equivalent synchronous* implementation. This implementation formally operates as a synchronous system, but, practically, does it in a looser fashion that makes it robust with respect to arbitrary, but discrete, latency variations between its processes.

A key element of a latency-insensitive protocols is the concept of *stalling event* (or, τ event), i.e. an event carrying as value the special symbol τ, as opposed to an *informative event*, i.e. an event carrying a value d in accordance with the original specification. A stalling events is the *modeling unit* to represent explicitly latency variations in inter-process communication, while remaining within the boundaries of the synchronous model. Hence, we augment domain $\mathcal{D}$ with τ, while Σ_{lat} denotes the set of all sequences of elements in $\mathcal{D} \ \cup \ \{\tau\}$. A *strict* signal s is always present and contains only informative events: $\forall t \in \mathcal{T}(s), (val(s,t) \notin \{\bot, \tau\})$. A *stalled* signal s contains at least one τ event: $(\exists t \in \mathcal{T}(s) \mid val(s,t) = \tau)$. Similarly to the definition of operator $\mathcal{F}_\bot$, we define operator $\mathcal{F}_\tau : \Sigma_{lat} \rightarrow \Sigma$ as follows:

$$\sigma'_i = \begin{cases} \sigma_{[t_i,t_i]}(s) \text{ if } \sigma_{[t_i,t_i]}(s) \neq \tau \\ \epsilon \qquad\qquad\quad \text{otherwise} \end{cases}$$

Two signals s, s' are *latency-equivalent* if they have the same sequence of values after discarding the τ events. This is written: $s \equiv_\tau s' \ \ \Leftrightarrow \ \ \mathcal{F}_\tau[\sigma(s)] = \mathcal{F}_\tau[\sigma(s')]$. As for semantic-equivalence, the definition of latency-equivalence extends to behaviors and processes. The main result of latency-insensitive design follows:

Theorem 1 ([10]). *If $S = \bigcap_i P_i$ is a strict synchronous system and $S' = \bigcap_i P'_i$ is a system of patient processes s.t. $\forall i \, (P'_i \equiv_\tau P_i)$ then $S' \equiv_\tau S$.*

Informally, a patient process is able to wait an arbitrary amount of reactions for an informative event to occur at any of its inputs and, when this occurs, it reacts as if not delay passed by. Hence, patience, which is compositional, enables the compensation of any arbitrary latency in inter-process communication. While patience is generally a strong requirement to demand, each *stallable* component (*core* process) can be made patient by generating proper interface logic (*shell* process) implementing the latency-insensitive protocol. A component is stallable if it can freeze its internal state for an arbitrary amount of time. In hardware systems, this property can be obtained through a *gated-clock mechanism*. At each reaction, the presence of a τ event at one of the input ports of a shell/core pair means that the expected informative event is not ready yet and will be delayed for at least another reaction. Hence, the shell logic reacts by stalling the internal core, while emitting new stalling events on the outputs and saving the informative events of its other input signals on internal buffers [3]. Once all the informative events for that reaction finally arrive, the interface reactivates the internal core, which progresses producing new informative events. It is important to notice that until *all* informative events for a given reaction arrive, the shell logic does *not* reactivate the core process, regardless of its internal computational structure. In fact, the shell logic ignores this structure and sees the

[3] To discuss the issue of how to determine the optimum size for these buffers or, alternatively, introduce *back-pressure* as part of the protocol goes beyond the scope of this paper (see [10,11]).

		t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
	tag :	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
P_l :	w_l :	1	0	τ	1	0	1	τ	0	...
	y_l :	0	2	2	τ	6	6	10	τ	...
	z_l :	0	τ	0	4	0	τ	8	4	...
Q_l :	w_l :	1	0	τ	1	0	1	τ	0	...
	x_{l_1} :	1	3	5	τ	7	9	11	τ	...
	y_l :	0	2	2	τ	6	6	10	τ	...
R_l :	w_l :	1	0	τ	1	0	1	τ	0	...
	x_{l_2} :	τ	1	3	5	τ	7	9	11	...
	z_l :	0	τ	0	4	0	τ	8	4	...

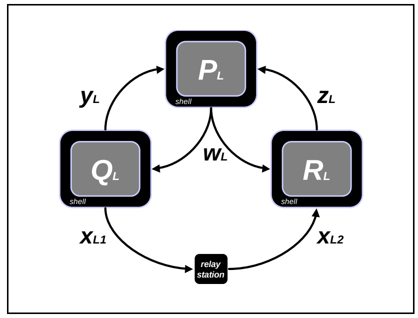

Fig. 4. The latency-insensitive system of Example 7 and its behavior.

core simply as a *block box* component. This conservative approach is a consequence of the assumption, influenced by single-clock hardware design, that the original system specification is strictly synchronous (a signal never presents a $\perp$ event at any reaction). In Section 4.4, we discuss how to lift this assumption to extend the application of latency-insensitive design to multi-clock and software systems.

Example 7. The application of latency-insensitive design to integrated circuits provides two main advantages: (a) it enables the *a-posteriori* automatic pipelining of long wires by insertion of special patient processes called *relay stations* [8]; (b) it facilitates the assembly of pre-designed components, that, as long as they are stallable, can be interfaced to the communication protocol without changing their internal structure. Assume that each process of Example 2 is implemented as a distinct finite state machine on an integrated circuit and that the wire carrying signal x from Q to R is the only one that has been pipelined by introducing one relay station. Figure 4 reports the structure and the behavior of the resulting latency-insensitive system after each process has been made patient by wrapping it within a shell. Note that the system is strictly synchronous (no absent-value $\perp$ events occur at any tag). At the system initialization, the only stalling event is the one at the output of the relay station because a relay station represents a "design correction" that is extraneous to the original system specification, while each finite state machine is properly initialized according to it. As the behavior evolves, new stalling events are generated whenever a process must wait for a new informative event at its inputs. In fact, since the system is cyclic [11], τ events occur periodically on each signal at the rate of $1/4$.

4.3 Desynchronization of Synchronous Programs.

The behavior of a synchronous system can be seen as a sequence of tuples of events with each tuple indexed by a tag. This is not the case for an asynchronous or a GALS system: the most one can say is that a behavior, being a tuple of signals, is a tuple of sequences of events. In [2,3,20], *desynchronization* is defined as the act of discarding the synchronization barriers that delimit successive reactions in a synchronous program. Since this corresponds to filtering away the absent-value events, desynchronization amounts to mapping a sequence of tuples of values in domains extended with the extra value $\perp$ into a tuple of sequences of present values, one sequence per each variable. The desynchronization abstraction is relevant because it provides a formal way to characterize those synchronous programs that can be deployed on a distributed architecture without losing their semantics. As proven in [3], the notions of endochrony and isochrony are sufficient for this characterization.

Let $clk(s,t)$ be a Boolean function denoting whether signal s presents an event at tag t or no, i.e. $(clk(s,t) = 1 \Leftrightarrow val(s,t) \neq \perp)$. For any process $P = (V, \mathcal{T}, \mathcal{B})$, any tag $t \in \mathcal{T}$, and any pair of subsets W, W' s.t. $W \subset W' \subseteq V$, we say that W *tag-determines* W' *at* t, written $W \rightarrow_t W'$, when:

$$\forall b \in \mathcal{B}, \forall s \in b_{|W'}, \Big(\exists \phi : D_W^t \rightarrow \{1,0\} \mid clk(s,t) = \phi(D_W^t) \Big)$$

where D_W^t is the set of values $val(s,t)$ for $s \in b_{|W}$. Since relation $W \rightarrow_t W'$ is stable by union over W' sets, there is a maximal W' that is tag-determined by W at t. Thus, for any tag $t \in \mathcal{T}$, if V is a finite set then there is a maximal chain $\emptyset = W_0 \rightarrow_t W_1 \rightarrow_t W_2 \rightarrow_t \ldots \rightarrow_t W_{max}$. A process $P = (V, \mathcal{T}, \mathcal{B})$ is *endochronous* when $\forall t \in \mathcal{T}$, $(W_{max} = V)$. In other words: a process is endochronous when for each tag of its behaviors the presence/absence of event on all its signals can be inferred incrementally from the values carried by a subset of them that are guaranteed to be present at this tag.

Two processes $P = (V, \mathcal{T}, \mathcal{B}), P' = (V', \mathcal{T}', \mathcal{B}')$ are *isochronous* when for each behaviors $b \in \mathcal{B}$ there is a behavior $b' \in \mathcal{B}'$ (and vice versa) s.t.:

$$\forall t \in \mathcal{T}, \; (W_t^\star \neq \emptyset \Rightarrow W_t^\star = W)$$

where $W = V \cap V'$ and $W_t^\star = \{s \in W \mid val(b,s,t) = val(b',s,t) \neq \perp\}$. In other words: two processes are isochronous when, at each tag, if there is a pair of shared signals that are present and agree on the event value, then for each other pair of shared signals, either they are present and agree on their value or they are absent.

Endochrony and isochrony allow the derivation of a key result for the automatic generation of distributed embedded code: *if each process of a synchronous* *program is endochronous and all processes are pairwise isochronous, then deploying the program on a GALS architecture gives a semantic-equivalent implementation* [3]. For any process $P(V, \mathcal{T}, \mathcal{B})$, let $\alpha(P)$ denote the semantic equivalent asynchronous process that is constructed from P by: (1) applying transformation $\mathcal{F}_{\perp}[\sigma(s)]$ to each signal $s \in b$, for all $b \in \mathcal{B}$, and (2) properly adding tags to each event of $\alpha(P)$ s.t. $\forall s, s' \in b, (\mathcal{T}(s) \cap \mathcal{T}(s') = \emptyset)$.

Theorem 2 ([3]). *Let $\bigcap_i P_i$ be a synchronous system such that each P_i is endochronous and each pair (P_i, P_j) is isochronous. Then $\bigcap_i \alpha(P_i) = \alpha(\bigcap_i P_i)$.*

Endochrony and isochrony can be expressed in terms of transition relations, are model checkable and synthesizable. Recall that the SIGNAL compiler handles the parallelism specified using synchronous composition by organizing the computation of signals as a collection of hierarchical trees (the *clock hierarchy forest*) based on the relations among their clocks. Hence, in practice, a synchronous program can be made endo-isochronous by adding a set of "Boolean guard" variables and a master clock to transform this forest into a tree. This approach is somewhat equivalent to synthesize an inter-process communication protocol and carries a drawback: there is not unique solution, or, in other words, there are many possible communication protocols.

4.4 Latency-Insensitive Design and Endo-Isochrony

Several commonalities between the work on synchronous program desynchronization and latency-insensitive design have been already pointed out in [1]. Here, we return on the topic to understand how the two approaches could be combined. First, we report a simple theoretical result.

Theorem 3. *The processes of a latency-insensitive system satisfy the properties of endochrony and isochrony.*

Proof. Let S be a latency-insensitive system. For each behavior of a process P of S, there is never a $\perp$ event on any signal. Thus, all signals of P belong to the same clock-equivalence class that coincides with the class of the activation clock of P and their presence is always trivially established. Hence, with respect to the above definition of endochrony, $W_{max} = V$ for all tags t. This proves endochrony of P. For the isochrony of any pair of processes P and P' of S, it is sufficient to notice that for each behavior of P either there exists a behavior of P' that agrees with P on the values of the events of all its signals, which are all present, or it doesn't exists (*tertium non datur*). $\square$

The previous result should not come as a surprise if we reflect on the intrinsic differences between τ events and $\perp$ events: the former is used to maintain the semblance of synchronous behavior while the latter represents the lack of it.

In other words, a τ event tells the process that "the awaited value is not ready yet", whereas a $\perp$ event says "there is no value to wait for". Hence, the τ mechanism accounts for the arbitrary latency of interprocess communication while enforcing a synchronous behavior for the distributed latency-insensitive system. Consequently, as illustrated by the diagram of Example 7, τ events never leave the systems (unless for the particular case of acyclic systems) and a price in performance may be ultimately paid [11]. Instead, one may wonder whether it is possible to derive an alternative endo-isochronous implementation of the system of Example 7 that doesn't rely on latency-insensitive design. In theory, this is certainly possible but in practice it may be challenging and not necessarily advantageous. This depends on the inner structure of each process in the system. In the particular case of Example 7 the result would be positive because the analysis of the causality dependencies among the events shows that the first two events of output signal z do not depend on the first event of input signal x_{l_2}, the delayed event (see also Example 3). Hence, the endo-isochronous implementation would be able to *absorb* the $\perp$ event (which it would see instead of the τ event seen in the corresponding latency-insensitive system) and the behavior would progress without further event absences.

5 Concluding Remarks

We used the tagged-signal model together with a simple "running example" to provide a comparative view of various design approaches: synchronous, asynchronous, GALS, latency-insensitive, and synchronous programming. In particular, we studied the interplay among the concepts of event absence, event sampling, and communication latency in modeling distributed heterogeneous design. Finally, we presented a new comparison of synchronous program desynchronization and latency-insensitive design. The main operational difference between latency-insensitive design and synchronous program desynchronization can be expressed as follows: the former knows how to handle *black box* processes but does not know how to analyze/exploit *white box* ones (that are treated uniformly as if they were black box processes); the latter does not know how to handle black box processes and must analyze the inner structure of each white box process in the system (as well as the properties of each communicating pair), but it is clever in exploiting the information resulting from this analysis. These reflections naturally lead to consider two new research avenues for extending the work on latency-insensitive design: (1) for single-clock systems, we could analyze each process that comes as a white box module and learn its input/output causality dependencies to see if they allow us to absorb some τ events at given states; (2) for multi-clock systems, we could combine the two approaches by applying hierarchically first the endo-isochronous analysis to handle $\perp$ events and, then, the latency-insensitive protocol mechanism to handle τ events.

Acknowledgments

The collaboration with Albert Benveniste and Paul Caspi has been essential in deriving the considerations made in this paper. We gratefully acknowledge their contributions to our on-going discussions on specification and implementation of distributed systems.

References

1. A. Benveniste. Some synchronization issues when designing embedded systems from componenents. In T.A. Henzinger and C.M. Kirsch, editors, *Embedded Software. Proc. of the First Intl. Workshop, EMSOFT 2001. Tahoe City, CA*, volume 2211 of *LNCS*, pages 32–49, Berlin, October 2001. Springer Verlag.
2. 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 *LNCS*, pages 162–177, Berlin, August 1999. Springer Verlag.
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, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In R. Alur and I. Lee, editors, *Embedded Software. Proc. of the Third Intl. Conference, EMSOFT 2003. Philadelphia, PA*, LNCS, Berlin, October 2003. Springer Verlag.
5. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Language Twelve Years Later. *Proc. of the IEEE*, 91(1):64–83, January 2003.
6. A. Benveniste and P. Le Guernic. Hybrid Dynamical Systems Theory and the Signal Language. *IEEE Transactions on Automatic Control*, 5:535–546, May 1990.
7. G. Berry. *The Foundations of Esterel*. MIT Press, 2000.
8. L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for "correct-by-construction" latency insensitive design. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 309–315. IEEE, November 1999.
9. L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and D. Peled, editors, *Proc. of the 11th Intl. Conf. on Computer-Aided Verification*, volume 1633, pages 123–133, Trento, Italy, July 1999. Springer Verlag.
10. L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.
11. L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proc. of the Design Automation Conf.*, pages 361–367. IEEE, June 2000.
12. L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in SOC design. *IEEE Micro*, 22(5):24–35, Sep-Oct 2002.

13. J. Cortadella, A. Kondratyev, L. Lavagno, , and C. Sotiriou. A concurrent model for desynchronization. In *Proc. Intl. Workshop on Logic Synthesis*, Laguna Beach, CA, May 2003.
14. S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal methods, validation and synthesis. *Proc. of the IEEE*, 85(3):266–290, March 1997.
15. A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software. Proc. of the Second Intl. Workshop, EMSOFT 2002. Grenoble, France*, volume 2491 of *LNCS*, Grenoble, France, October 2002. Springer-Verlag.
16. P. Le Guernic, J. P. Talpin, and J. C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, April 2003.
17. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers", 1993.
18. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.
19. E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.
20. J. P. Talpin, A. Benveniste, B. Caillaud, and P. Le Guernic. Hierachical Normal Form for Desynchronization. Technical Report 3822, IRISA, 1999.
21. J. P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. A polychronous model for high-level component-based system design. In *Proc. European Design and Test Conf.*, 2003.
22. J. P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. Polychrony for formal refinement-checking in a system-level design methodology. In *Application of Concurrency to System Design (ACSD'2003)*, 2003.