# Loosely Time-Triggered Architectures: Improvements and Comparisons

Guillaume Baudart
DI École normale supérieure
Inria, Paris - Rocquencourt

Albert Benveniste
Inria, Rennes

Timothy Bourke
Inria, Paris - Rocquencourt
DI École normale supérieure

## ABSTRACT

Loosely Time-Triggered Architectures (LTTAs) are a proposal for constructing distributed embedded control systems. They build on the quasi-periodic architecture, where computing units execute 'almost periodically', by adding a thin layer of middleware that facilitates the implementation of synchronous applications.

In this paper, we show how the deployment of a synchronous application on a quasi-periodic architecture can be modeled using a synchronous formalism. Then we detail two protocols, *Back-Pressure* LTTA, reminiscent of elastic circuits, and *Time-Based* LTTA, based on waiting. Compared to previous work, we present controller models that can be compiled for execution and a simplified version of the Time-Based protocol. We also compare the LTTA approach with architectures based on clock synchronization.

## 1. INTRODUCTION

This paper is about implementing programs expressed as stream equations, like those written in Lustre, Signal, or the discrete subset of Simulink, over networks of embedded controllers. Since each controller is activated on its own local clock, some *middleware* is needed to ensure the correct execution of the original program. One possibility is to rely on a clock synchronization protocol as in the Time-Triggered Architecture (TTA) [21]. Another is to use less constraining protocols as in the Loosely Time-Triggered Architecture (LTTA) [2, 3, 5, 11, 26].

The embedded applications that we consider involve both continuous control and discrete logic. Since the continuous layers are naturally robust to sampling artifacts, controllers can simply communicate through shared memory without additional synchronization. But the discrete logic for mode changes and similar functionalities is very sensitive to such artifacts, and requires more careful coordination. The LTTA protocols are intended for this class of embedded systems. They extend communication by sampling with minimal mechanisms that preserve the semantics of the discrete layer. They are simple to implement and involve little additional network communication. They thus remain an interesting alternative to solutions based on clock synchronization despite their undeniable advantages.

There are two LTTA protocols: *Back-Pressure* and *Time-Based*. The Back-Pressure protocol is based on acknowledging the receipt of messages. While very efficient, it introduces control dependencies. The Time-Based protocol is based on a waiting mechanism. It is less efficient but allows controllers to operate more independently.

### *Contributions.*

In this paper we consolidate previous work on LTTAs [2, 11, 26] in a synchronous formalism that uniformly encompasses both protocols and applications. Indeed, protocol controllers are also synchronous programs: they can be compiled together with application code. Any synchronous language [4] could be used to express the general LTTA framework, its instantiations with specific protocols, and applications themselves. But we choose Zélus [6][1] because it also supports a continuous model of time, which allows the direct expression of timing constraints from the underlying network architecture, giving a single, coherent, and precise model. These timing constraints arise from the fact that controllers are activated *quasi-periodically*, that is periodically but with jitter, and because transmission delays are bounded. Not only do we clarify the models and reasoning presented in previous papers, but we give a simpler version of the Time-Based protocol and prove it correct. Finally, modern clock synchronization protocols are now cost-effective and precise [13, 21, 22, 25], raising the question: *Is there really any need for the LTTA protocols?* We thus compare, for the first time, the LTTA protocols with approaches based on clock synchronization.

### *Overview.*

In section 2, we formalize quasi-periodic architectures, model their timing constraints in Zélus, and recall the fundamentals of synchronous applications. Then, in section 3, we present a general framework for modeling controller networks and LTTA protocols. This framework is instantiated with the two LTTA protocols in section 4. Finally, in section 5, we compare the protocols to an approach based on clock synchronization.

*EMSOFT* '15 Amsterdam, Netherlands

.

[1]Appendix A presents an overview of Zélus. Appendix F presents the source code of a complete example.

## 2. WHAT IS AN LTTA?

An LTTA is the combination of a quasi-periodic architecture with a protocol for deploying synchronous applications. We now present the key definitions of quasi-periodic architectures (section 2.1) and synchronous applications (section 2.3).

### 2.1 Quasi-Periodic Architectures

Introduced in [10], the *quasi-synchronous approach* is a set of techniques for building distributed control systems. It is a formalization of practices that Paul Caspi observed while consulting in the 1990s at Airbus, where engineers were deploying synchronous Lustre/SCADE [16, 18] designs onto networks of non-synchronized nodes communicating via shared memories with bounded transmission delays.

The quasi-synchronous approach applies to systems of periodically executed (sample-driven) nodes. In contrast to the Time-Triggered Architecture [21], it does not rely on clock synchronization. Such systems arise naturally as soon as two or more microcontrollers running periodic tasks are interconnected. They are common in aerospace, nuclear power, and rail transportation.

Definition 1 (Quasi-periodic Architecture). *A* quasi-periodic architecture *is a finite set of nodes $\mathcal{N}$, where every node $n \in \mathcal{N}$ executes periodically but the actual time between any two activations $T \in \mathbb{R}$ may vary between known bounds during an execution:*

$$0 < T_{min} \leq T \leq T_{max}. \quad (1)$$

*Values are transmitted between processes with a delay $\tau \in \mathbb{R}$, bounded by $\tau_{min}$ and $\tau_{max}$,*

$$0 < \tau_{min} \leq \tau \leq \tau_{max}. \quad (2)$$

*Each is buffered at receivers until a newer value is received.*

Since we consider all possible behaviors, a quasi-periodic system can also be characterized by its nominal period $T_n$ and maximum jitter $\varepsilon$, where $T_{\min} = T_n - \varepsilon$ and $T_{\max} = T_n + \varepsilon$ and similarly for the transmission delay. The margins encompass all sources of divergence between nominal and actual values, including relative clock jitter, interrupt latencies, and scheduling delays. We assume that individual processes are synchronous: reactions triggered by a local clock execute in zero time (atomically with respect to the local environment).

In the original quasi-synchronous approach, transmission delays are only constrained to be 'significantly shorter than the periods of read and write clocks' [10, § 3.2.1]. We introduce explicit bounds in equation (2) to make the definition more precise and applicable to a wider class of systems. They can be treated naturally in our modeling approach.

Nodes communicate through shared memories which are updated atomically. A given variable is updated by a single node, but may be read by several nodes. The values written to a variable are sent from the producer to all consumers, where they are stored in a specific (one-place) buffer. The buffer is only sampled when the process at a node is activated by the local clock. This model is sometimes termed *Communication by Sampling* (CbS) [3].

Finally, we assume that the network guarantees message delivery and preserves message order. That is, for the latter, if message $m_1$ is sent before $m_2$, then $m_2$ is never received before $m_1$. This is necessarily the case when $\tau_{\max} < T_{\min} + \tau_{\min}$, otherwise this assumption only burdens implementations with the technicality of numbering messages and dropping those that arrive out of sequence.

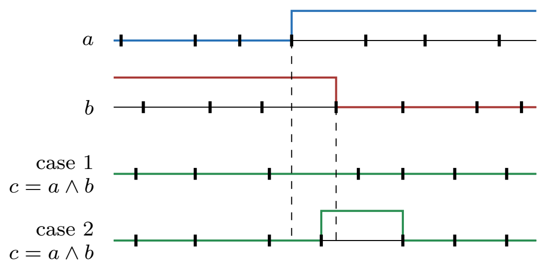


Figure 1: The effect of sampling on signal combinations.

#### *Value duplication and loss.*

The lack of synchronization in the quasi-periodic architecture means that successive variable values may be duplicated or lost. For instance if a consumer of a variable is activated twice between the arrivals of two successive messages from the producer, it will *oversample* the buffered value. On the other hand, if two messages of the producer are received between two activations of the consumer, the second value *overwrites* the first, which is then never read. These effects occur for any $\varepsilon > 0$, regardless of how small.

The timing bounds of definition 1 mean, however, that the maximum numbers of consecutive oversamplings and overwritings are functions of the bounds on node periods and transmission delays (see appendix B for proofs).

Property 1. *Given a pair of nodes executing and communicating according to definition 1, the maximum number of consecutive oversamplings and overwritings is*

$$n_{os} = n_{ow} = \left\lceil \frac{T_{max} + \tau_{max} - \tau_{min}}{T_{min}} \right\rceil - 1. \quad (3)$$

This property implies that data loss can be prevented by activating a consumer much more frequently than the corresponding producer (at the cost of higher oversampling). Quasi-periodic architectures involving producer-consumer pairs are studied in [5].

Quasi-periodic architectures are a natural fit for continuous control applications, where the error due to sampling artifacts can be computed and compensated. In this paper, however, we treat discrete systems, like state machines, which are generally intolerant to data duplication and loss.

#### *Signal combinations.*

There is another obstacle to implementing discrete applications on a quasi-periodic architecture: naively combining variables can give results that diverge from the reference semantics. Consider, for example, figure 1 [10, §4.2.2][2, 11]. A node $C$ reads two boolean inputs $a$ and $b$, produced by nodes $A$ and $B$, respectively, and computes the conjunction, $c = a \wedge b$. Here, $a$ is *false* for four activations of $A$ before becoming *true* and $b$ is *true* for four activations of $B$ before becoming *false*. In a synchronous semantics, with simultaneous activations of $A$, $B$ and $C$, node $C$ should return *false* at each activation. But, as figure 1 shows, the value computed depends on when each of the nodes is activated. This phenomena cannot be avoided by activating nodes less or more frequently.

## 2.2 Modeling Quasi-Periodic Architectures

One of the central ideas of the original quasi-synchronous approach is to replace detailed timing behavior with a discrete abstraction [10, §3.2]. Basically, a system is modeled, in Lustre, for example, as a composition of discrete programs activated by a 'scheduler' program that limits interleaving [19]. Now, rather than arising as a consequence of the timing constraints of definition 1, properties like property 1 are enforced directly by the scheduler. This approach allows the application of discrete languages, simulators, and model-checkers, but it does not apply to the present setting where 'short undetermined transition delays' [10, §3.2.1] are replaced by equation (2). In fact, Caspi knew that 'if longer transmission delays are needed, modeling should be more complex' [10, §3.2.1, footnote 2]. The earliest paper on LTTAs [5] models messages in transmission, but still in a discrete model. Later papers introduce a class of protocols that rely on the timing behavior of the underlying architecture. Their models mix architectural timing constraints with protocol details using automata [11] or ad hoc extensions of timed Petri nets [2]. In contrast, we use Zélus [6], a synchronous language extended with continuous time, where we can clearly separate real-time constraints from discrete control logic, but still combine both in an executable language.

Let us first consider a quasi-periodic clock that triggers the activation of an LTTA node according to equation (1). Such a clock can be simulated in Zélus using a timer, a simple Ordinary Differential Equation (ODE) $\dot{t} = 1$, initialized to an arbitrary value between $-T_{\min}$ and $-T_{\max}$, and similarly reinitialized whenever $t$ reaches 0. As Zélus is oriented towards numerical simulation, we must make two compromises for our program to be executable. First, rather than an arbitrary value, we choose at random:[2]

```
let node arbitrary (l, u) = l +. Random.float (u −. l)
```

This declares a function named arbitrary with two inputs and defined by a single expression; the keyword **node** indicates a discrete stream function. Second, the reinitialization condition is encoded as a *(rising) zero-crossing expression* which a numeric solver monitors to detect and locate significant instants. These choices made, the model for node clocks is:

```
let hybrid metro (t_min, t_max) = c where
  rec der t = 1.0 init −. arbitrary (t_min, t_max)
          reset up(last t) → −. arbitrary (t_min, t_max)
  and present up(last t) → do emit c = () done
```

*val metro : float ∗ float* $\xrightarrow{\mathrm{C}}$ *unit signal*

The keyword **hybrid** indicates that node inputs and outputs are continuous. The variable t is initialized as described above and increases with slope 1.0. The expression **up**(**last** t) registers a zero-crossing expression on (the left-limit of) t. At zero-crossing instants, a signal c is emitted and t is reset.

Similarly, the constraint on transmission delays from equation (2) is modeled by delaying the discrete signal corresponding to the sender's clock. A simple Zélus model is:

```
let hybrid delay(c, tau_min, tau_max) = dc where
  rec der t = 1.0 init 0.0
          reset c() →  −. arbitrary  (tau_min, tau_max)
  and present up(t) → do emit dc = () done
```

*val delay : unit signal ∗ float ∗ float* $\xrightarrow{\mathrm{C}}$ *unit signal*

[2]+. , −., ∗., /. denote floating-point operations

The function delay takes a clock c as input. When c ticks, the timer is reinitialized to an arbitrary value between $-\tau_{\min}$ and $-\tau_{\max}$ corresponding to the transmission delay. Then, when the delay has elapsed, that is, when a zero-crossing is detected, a signal dc for the delayed clock is emitted. The presented model is simplified for readability. In particular, it does not allow for simultaneous ongoing transmissions, that is, it mandates $\tau_{\max} < T_{\min}$. The full version, given in appendix D, queues ongoing transmissions which complicates the model without providing any new insights.

## 2.3 Synchronous Applications

This paper addresses the deployment of *synchronous applications* onto a quasi-periodic architecture. By synchronous application, we mean a synchronous program that has been compiled into a composition of communicating Mealy machines. The question of generating such a form from a high-level language like Lustre/SCADE, Signal, or Esterel [4] does not concern us here.

In the synchronous model, machines are executed in lock-step. But as our intent is to distribute each machine onto its own network node, we must show that a desynchronized execution yields the same overall input/output relation as the reference semantics. The aim is to precisely describe the activation model and the related requirements on communications, and thereby the form of, and the constraints on program distribution. The desynchronized executions we consider are still idealized—reproducing them on systems satisfying definition 1 is the subject of section 4.

A Mealy machine $m$ is a tuple $\langle s_{\text{init}}, I, O, F\rangle$, where $s_{\text{init}}$ is an initial state, $I$ is a set of input variables, $O$ is a set of output variables, and $F$ is a transition function mapping a state and input values to the next state and output values:

$$F \;:\; \mathcal{S} \times \mathcal{V}^I \to \mathcal{S} \times \mathcal{V}^O$$

where $\mathcal{V}$ is the domain of variable values, and $\mathcal{S}$ is the domain of state values. A Mealy machine $m = \langle s_{\text{init}}, I, O, F\rangle$ defines a stream function[3]

$$[\![m]\!] \;:\; (\mathcal{V}^I)^\infty \to (\mathcal{V}^O)^\infty$$

generated by repeated firings of the transition function from the initial state:

$$\begin{aligned} &s(0) = s_{\text{init}} \\ &s(n+1),\; o(n) = F(s(n), i(n)). \end{aligned}$$

The fact that the outputs of Mealy machines may depend instantaneously on their inputs makes both composition [24] and distribution over a network [12] problematic. An alternative is to only consider a 'Moore-style' composition of Mealy machines: outputs may be instantaneous but communications between machines must be delayed. A machine must wait one step before consuming a value sent by another machine. This choice precludes the separation of subprograms that communicate instantaneously, but it increases node independence and permits simpler protocols. For a variable $x$, let ${}^\bullet x$ denote its delayed counterpart (${}^\bullet x(n) = x(n-1)$). Similarly, let ${}^\bullet X = \{{}^\bullet x \mid x \in X\}$. Now, a set of machines $m_1, m_2, \ldots, m_p$ can be composed to form a *system* $N = m_1 \mid\mid m_2 \mid\mid \ldots \mid\mid m_p$. The corresponding

[3]$\mathcal{X}^\infty = \mathcal{X}^* \cup \mathcal{X}^\omega$ denotes the set of possibly finite streams over elements of the set $\mathcal{X}$.

Mealy machine $N = \langle s_{\text{init}}, I, O, F_N \rangle$ is defined by

$$
\begin{aligned}
&I = I_1 \cup \cdots \cup I_p \setminus {}^\bullet O, \\
&O = O_1 \cup \cdots \cup O_p, \\
&s_{\text{init}} = (s_{\text{init}_1}, \ldots, s_{\text{init}_p}, nil, \ldots, nil) \\
&F_N((s_1, \ldots, s_p, {}^\bullet O),\ I) = ((s'_1, \ldots, s'_p, O),\ O)
\end{aligned}
$$

where $(s'_i, o_i) = F_i(s_i, i_i)$. The actual inputs of the global Mealy machine are the inputs of all machines $m_i$ that are not delayed versions of variables produced by other machines. At each step a delayed version of the output of machines $m_i$, initialized with *nil*, is stored into the state of the global Mealy machine. The abuse of notation in $F_N$ describes the shuffling of input, output, and delayed variables.

The composition is well defined if the following conditions hold: For all $m_i \neq m_j$,

$$I_i \cap O_j = \emptyset, \tag{4}$$
$$O_i \cap O_j = \emptyset, \text{ and} \tag{5}$$
$$I_i \setminus {}^\bullet O \cap I_j \setminus {}^\bullet O = \emptyset, \tag{6}$$

Equation (4) states that no machine ever directly depends on the output of another. Equation (5) imposes that a variable is only defined by one machine. Finally, equation (6) states that an input from the environment is only consumed by a single machine. Additionally, since the delayed outputs are initially undefined, the composition is only well defined when the $F_i$ do not depend on them at the initial instant.

In the synchronous model, all processes run in lock-step, that is, executing one step of $N$ executes one step of each $m_i$ in no particular order. Thus, at each step, all inputs are consumed simultaneously to immediately produce all outputs. The *Kahn semantics* [20] proposes an alternative model where each machine is considered a function from a tuple of input streams to a tuple of output streams (the variables effectively become unbounded queues). Synchronization between distinct components of tuples and between the activations of elements in a composition are no longer required. The semantics of a program is defined by the sequence of values at each variable:

$$[\![m]\!]^K \ :\ (\mathcal{V}^\infty)^I \to (\mathcal{V}^\infty)^O.$$

Theorem 1. *For Mealy machines, composed as described above, the synchronous semantics and the Kahn semantics are equivalent*

$$[\![m]\!] \ \approx [\![m]\!]^K.$$

The proof of this theorem is given in appendix C.

The overall idea is to take a synchronous application that has been arranged into a Moore-composition of Mealy machines $N = m_1 \mid\mid m_2 \mid\mid \ldots \mid\mid m_p$, so that each machine $m_i$ can be placed on a distinct network node. If the transmission and consumption of values respects the Kahn semantics then the network correctly implements the application. Since we do not permit instantaneous dependencies between variables computed at different nodes, a variable x computed at one node may only be accessed at another node through a *unit delay*, that is, a delay of one logical step. In this way we need not 'microschedule' node activations.

## 3. GENERAL FRAMEWORK

We now consider the implementation of a synchronous application $S$ of $p$ Mealy machines communicating through unit delays onto a quasi-periodic architecture with $p$ nodes.

This task is trivial if the underlying nodes and network are *completely synchronous*, that is, $T_{\min} = T_{\max} \geq \tau_{\max}$ and with all elements initialized simultaneously. One simply compiles each machine and assigns it to a node. At each tick, all the machines compute simultaneously and send values to be buffered at consumers for use at the next tick. The synchronous semantics of an application is preserved directly.

In our setting, however, node activations are not synchronized and we must confront the artifacts described in section 2.1: duplication, loss of data, and potentially unintended signal combinations. We do this by introducing a thin layer of middleware between application and architecture. An LTTA is exactly this combination of a quasi-periodic architecture with a protocol that preserves the semantics of synchronous applications. We denote the implementation of an application $S$ on a quasi-periodic architecture as LTTA($S$). In this section we present the general framework of this implementation based on a discrete synchronous model of the architecture. The details of the two LTTA protocols are presented in section 4.

### 3.1 From Continuous to Discrete Time

We describe the protocols by adapting a classical approach to architecture modeling using synchronous languages [17]. In doing so, we exploit the ability of the Zélus language [6] to express delays without a priori discretization.

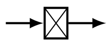
The quasi-periodic architecture is modeled by a set of clocks (see section 2.2). Signals c1, c2, ... denote the quasi-periodic clocks of the nodes, and dc1, dc2, ... their delayed versions that model transmission delays. The union of all these signals is a global signal g which is emitted on each event. In Zélus, we write:

```
present c1() | dc1() | c2() | dc2() | ... → do emit g = () done
```

The signal g gives a base notion of logical instant or step. It allows us to model the rest of the architecture in a discrete synchronous framework.

Variables are not necessarily always defined. This is expressed in Zélus by valued *signals*. If necessary, a signal value can be maintained in a *memory* which stores the last received value until the next update.

```
let node mem(i, default) = m where
  rec init m = default
  and present i(v) → do m = v done
```



*val mem : 'a signal ∗ 'a* $\xrightarrow{\text{D}}$ *'a*

The keyword init initializes a memory m with a default value default. Each time the input signal i is emitted, the variable m is updated with the new received value v.

### 3.2 Modeling Nodes

An LTTA node is formed by composing a Mealy machine with a controller that determines when to execute the machine and when to send outputs to other nodes. The basic idea comes from the *shell wrappers* of Latency Insensitive Design (LID) [8,9]. The schema is shown in figure 2. A node is activated at each tick of its quasi-periodic clock c:

```
present c() → do o = ltta_node(i, default) done
```

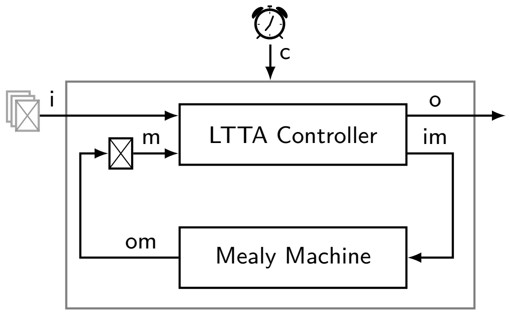


Figure 2: Schema of an LTTA node: a Mealy machine is encapsulated with a protocol controller. The crossed box is implemented by the mem function defined in section 3.1.

An LTTA node is modeled in Zélus as:

```
let node ltta_node(i, default) = o where
  rec m = mem(om, default)
  and (o, im) = ltta_controller(i, m)
  and present im(v) → do emit om = machine(v) done
```

*val ltta_node : 'a list ∗ 'b* $\xrightarrow{D}$ *'b signal*

The controller node is instantiated with one of the controllers described in the following section. At instants determined by the protocol, the controller samples a list of inputs from incoming LTTA links i and passes them on im to trigger the machine, which produces output om. The value of om is stored in a memory m, and sent on outgoing LTTA links o when the protocol allows.

The function of the controller is to preserve the semantics of the global synchronous application by choosing 1) when to execute the machine (emission of signal im), and, 2) when to send the resulting outputs (emission of signal o). Neither of the two presented protocols computes and sends an output instantaneously. They both thus reintroduce the unit delays required for correct distribution.

## 3.3 Modeling Links

Delayed communications are modeled by an unbounded FIFO queue that is triggered by the input signal and the delayed sender clock that models transmission delays dc (see section 2.2). Messages in transmission are stored in the queue and emitted when the transmission delay elapses, that is, if clock dc ticks when the queue is not empty.

```
let node channel(dc, i) = o where
  rec init q = empty()
  and trans = not (is_empty (last q))
  and present
      | i(v) & dc() on trans →
          do emit o = front(last q)
          and q = enqueue(dequeue(last q), v) done
      | i(v) → do q = enqueue(last q, v) done
      | dc() on trans →
          do emit o = front(last q)
          and q = dequeue(last q) done
```

*val channel : unit signal ∗ 'a signal* $\xrightarrow{D}$ *'a signal*

Each new message v received on signal i is added at the end of the queue q: q = enqueue(**last** q, v). The keyword **last** refers to the last defined value of a variable. Then, when a transmission delay has elapsed, that is, each time clock dc

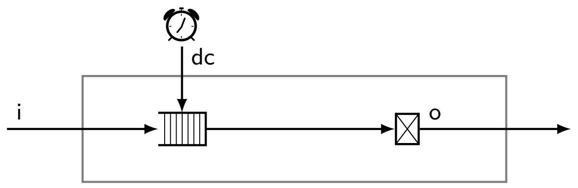


Figure 3: Schema of communication links modeling delayed transmission between nodes. The striped box represents a FIFO queue.

ticks when the queue is not empty (when trans is set to *true*), the first pending message is emitted on signal o and removed from the queue: **emit** o = front(**last** q) and q = dequeue(**last** q).

Finally, a link between two distinct nodes, shown in figure 3, stores the last received value in a memory. Since nodes are not synchronized, the output of a link must be defined at each logical step. All link nodes are thus activated at every emission of g.

```
let node link(dc, i, default) = o where
  rec s = channel(dc, i)
  and o = mem(s, default)
```

*val link : unit signal ∗ 'a signal ∗ 'a* $\xrightarrow{D}$ *'a*

When a message is sent on signal i, it goes through the channel and, after the transmission delay, is stored in a memory. New messages overwrite previous memory values. The memory contents are output by the link.

### *Fresh values.*

The LTTA controllers must detect when a fresh write is received in an attached shared memory even when the same value is resent. An *alternating bit* protocol suffices for this task since the controllers ensure that no values are missed:

```
type 'a msg = {data : 'a; alt : bool}
let node alternate i = o where
  rec present i(v) → local flag in
        do flag = true → not (pre flag)
        and  emit o = {data = v; alt = flag} done
```

*val alternate : 'a signal* $\xrightarrow{D}$ *'a msg signal*

The value of the boolean variable flag is associated to each new value received on signal i. This value alternates between *true* and *false* at each emission of signal i. This simple protocol logic is readily incorporated into the link model.

```
let node ltta_link(dc, i, default) = o where
  rec s = channel(dc, i)
  and o = mem(alternate(s), default)
```

*val ltta_link : unit signal ∗ 'a signal ∗ 'a msg* $\xrightarrow{D}$ *'a msg*

An alternating bit is associated to each new value stored in the memory. Within a controller, the freshness of an incoming value can now be detected and signaled:

```
let node fresh (i, r) = o where
  rec init m = false
  and present r(_) → do m = i.alt done
  and o = (i.alt <> last m)
```

*val fresh : 'a msg ∗ 'b signal* $\xrightarrow{D}$ *bool*

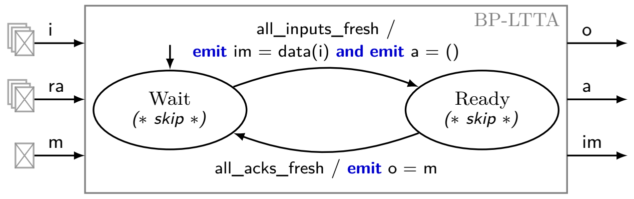


Figure 4: The Back-Pressure LTTA controller. The additional inputs ra are acknowledgments from consumers. The additional output a is for acknowledging producers.

Variable m stores the alternating bit associated to the last read value and is updated at each new read signaled by an emission on r. A fresh value is detected when the current value of the alternating bit differs from the one stored in m, that is, (i.alt <> **last** m).

## 4. THE LTTA PROTOCOLS

We now present two LTTA protocols: one based on back-pressure (section 4.1) and another based on time (section 4.2).

### 4.1 Back-Pressure LTTA

The Back-Pressure protocol [26] is inspired by *elastic circuits* [14, 15] where a consumer node must acknowledge each value read by writing to a *back pressure* [7] link connected to the producer. This mechanism allows executing a synchronous application on an asynchronous architecture while preserving the Kahn semantics. In an elastic circuit nodes are triggered as soon as all their inputs are available. This does not work for LTTA nodes since they are triggered by local clocks, so a *skipping* mechanism was introduced in [26] and included in later Petri net formalizations [1, 2].

For each link from a node $A$ to a node $B$, we introduce a back-pressure link from $B$ to $A$. This link is called a (acknowledge) at $B$ and ra (receive acknowledge) at $A$. The controller, shown in figure 4, is readily programmed in Zélus:

```
let node bp_controller (i, ra, m) = (o, a, im) where
  rec automaton
      | Wait →
         do (∗ skip ∗)
         unless all_inputs_fresh then
           do emit im = data(i) and emit a = () in Ready
      | Ready →
         do (∗ skip ∗)
         unless all_acks_fresh then
           do emit o = m in Wait

  and all_inputs_fresh = forall_fresh(i, im)
  and all_acks_fresh = forall_fresh(ra, o)
```

*val bp_controller :*
*'a msg list ∗ 'b msg list ∗ 'c* $\xrightarrow{\text{D}}$
*'c signal ∗ unit signal ∗ 'a list signal*

The controller automaton has two states. It starts in Wait and skips at each tick until fresh values have been received on all inputs. It then triggers the machine (data() discards the alternating bit), sends an acknowledgement to the producer and transitions immediately to Ready. The controller skips in Ready until acknowledgements have been received from all consumers indicating that they have consumed the most recently sent outputs. It then sends the outputs from the last activation of the machine and returns to Wait.

In parallel, the freshness of the inputs since the last execution of the machine is tested by a conjunction of fresh nodes. Similarly the controller also tests whether fresh acknowledgments have been received from all consumers since the last emission of the output signal o.

THEOREM 2. *The composition of a Back-Pressure controller and a Mealy machine to form a Back-Pressure LTTA node is well defined.*

PROOF. The dependency graph of the controller is:

$$\mathsf{im} \leftarrow \mathsf{i} \quad \mathsf{a} \leftarrow \mathsf{i} \quad \mathsf{o} \leftarrow \mathsf{ra} \quad \mathsf{o} \leftarrow \mathsf{m}.$$

The definition of the local memory m adds the dependency m ← om. Since the dependency graph of the machine is, at worst, om ← im, the composition of the two machines is free of cycles and therefore well defined. □

### *Preservation of Semantics.*

This result was first proved in [26] for networks of nodes communicating through buffers of arbitrary size. Another proof is given in [2] based on the relation with elastic circuits. We sketch this proof here; details are to be found in [1].

THEOREM 3 ([2, 26]). *Implementing a synchronous application $S$ over a quasi-periodic architecture with Back-Pressure controllers preserves the Kahn semantics of the application:*

$$[\![LTTA_{\text{BP}}(S)]\!]^K = [\![S]\!]^K.$$

PROOF. The idea is to show that both nodes and links can be encoded as simple event graphs, that is, one-safe Petri nets, that behave like elastic circuits. Event graphs associated to links contain two places, a direct place that models data transfer and a back-pressure place that models consumer acknowledgment. The controller is also modeled as an event-graph with two places that alternate between *send* and *execute* transitions. This net is then extended with a skipping mechanism that allows the controller to transition on local clock ticks if not all inputs are available.

Now, assuming that no transition can be enabled forever without firing, one can show that the LTTA implementation of the application behaves like the elastic circuit version of the same application which is known to implement the Kahn semantics of a synchronous application. □

### *Performance Bounds.*

Using the assumptions on local clock periods and transmission delays of the underlying quasi-periodic architecture, that is, equations (1) and (2), we can analyze the performance of Back-Pressure LTTA nodes.

THEOREM 4 ([2, 26]). *The worst case throughput of a Back-Pressure LTTA node is*

$$\lambda_{\text{BP}} = 1/2(T_{max} + \tau_{max}).$$

A formal proof based on Petri nets is presented in [1]. Here we give just an intuition for the case of two nodes $A$ and $B$ with $B$ receiving messages from $A$. In the worst case, when $A$ sends a message at time $t$ it arrives in $B$'s shared memory at $t + \tau_{\max}$ just after a tick of $B$'s clock. Therefore $B$ does

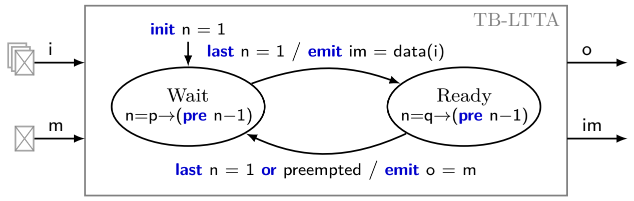

TB-LTTA
init n = 1
last n = 1 / emit im = data(i)
i
m
o
im
Wait
n=p→(pre n−1)
Ready
n=q→(pre n−1)
last n = 1 or preempted / emit o = m

Figure 5: The Time-Based LTTA controller. A counter n is decremented in each state initialized with value p in state WAIT and q in state READY.

not detect the message until $t + T_{\max} + \tau_{\max}$.[4] Symmetrically, the acknowledgment from $B$ to $A$ is received $T_{\max} + \tau_{\max}$ after the message has been detected. The next round thus starts at $t + 2(T_{\max} + \tau_{\max})$.

## 4.2 Time-Based LTTA

The Time-Based LTTA protocol realizes a synchronous execution on a quasi-periodic architecture by alternating *send* and *execute* phases across the nodes. Each node maintains a local countdown whose initial value is tuned according to the timing characteristics of the architecture to ensure that, when the countdown elapses, it is safe to execute the machine or publish its most recent results.

A first version of the Time-Based LTTA protocol was introduced in [10]. The protocol was formalized as a Mealy machine with five states in [11] and a simplified version was modeled with Petri nets in [1,2]. We propose an even simpler version, formalize it in Zélus, and prove its correctness.

Unlike the Back-Pressure protocol, the Time-Based protocol requires *broadcast communication*, that is, all variable updates must be visible at all nodes (and each node must update at least one variable), but acknowledgment values are not sent when inputs are sampled. The controller for the Time-Based protocol is shown in figure 5, for parameters p and q:

```
let node tb_controller (i, ro, m) = (o, im) where
  rec init n = 1
  and automaton
      | Wait →
          do n = p → (pre n − 1)
          unless (last n = 1) then
            do emit im = data(i) in Ready
      | Ready →
          do n = q → (pre n − 1)
          unless ((last n = 1) or preempted) then
            do emit o = m in Wait

  and preempted = exists_fresh(i, im)

val tb_controller :
  'a msg list ∗ 'b msg list ∗ 'c -D-> 'c signal ∗ 'a list signal
```

The controller automaton has two states. Initially, it passes via Wait, emits the signal im with the value of the input memory i and thereby *executes* the machine, and enters Ready. In Ready, the equation n = q → (pre n − 1) initializes a counter n with the value q and decrements it at each subsequent tick of the clock c. At the instant when the Ready counter would become zero, that is, when the previous value last n is one, the controller instead passes directly into the Wait state, resets the counter to p, and *sends* the previously computed outputs from the memory m (see figure 2) to o. It may happen, however, that the local clock is much slower than those of other nodes. In this case, a fresh value from any node, exists_fresh(i, im), preempts the normal countdown and triggers the transition to Wait and the associated writing of outputs (exists_fresh is essentially a disjunction of fresh nodes). The Wait state counts down from p to give all inputs enough time to arrive before the machine is retriggered.

Basically, nodes slow down by counting to accomodate the unsynchronized activations of other nodes and message transmission delays, but accelerate when they detect a message from other nodes.

THEOREM 5. *The composition of a Time-Based controller and a Mealy machine to form a Time-Based LTTA node is always well defined.*

PROOF. The proof is similar to that of theorem 2. The worst case dependency graph of a node is:

$$n \leftarrow i \quad o \leftarrow m \quad m \leftarrow om \quad om \leftarrow im \quad im \leftarrow i.$$

It has no cyclic dependencies. □

### *Preservation of Semantics.*

The Time-Based protocol only preserves the Kahn semantics of the application if the countdown values $p$ and $q$ are correctly chosen.

THEOREM 6. *The Kahn semantics of a synchronous application $S$ implemented on a quasi-periodic architecture using Time-Based controllers is preserved,*

$$[\![LTTA_{\text{TB}}(S)]\!]^K = [\![S]\!]^K$$

*provided that both*

$$p > \frac{2\tau_{max} + T_{max}}{T_{min}} \tag{7}$$

$$q > \frac{\tau_{max} - \tau_{min} + (p+1)T_{max}}{T_{min}} - p. \tag{8}$$

PROOF. The theorem follows from two properties which together imply that the $k$th execution of a node samples the $(k-1)$th values of its producers. Since nodes communicate through unit delays, the Kahn semantics is preserved.

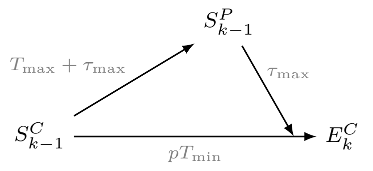
PROPERTY 2 $(S^P_{k-1} \prec E^C_k)$. *For $k > 0$, the $(k-1)$th sending of a producer is received at its consumers before their respective $k$th executions.*

PROPERTY 3 $(E^C_k \prec S^P_k)$. *For $k > 0$, the $k$th execution of a consumer occurs before the $k$th sending from any of its producers is received.*

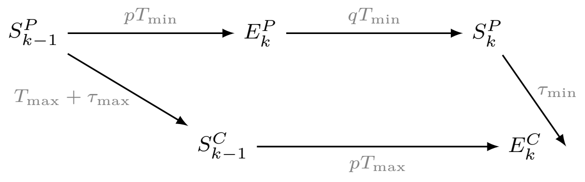
The properties are shown by induction on $k$; that is, assuming that they hold up to and including $k-1$. The proofs proceed by considering the worst-case scenarios illustrated in Figure 6.

For property 2, if the $k$th execution of a consumer $E^C_k$ occurs at time $t$ then its $(k-1)$th sending $S^C_{k-1}$ must have occurred at or before $t - pT_{\min}$ (countdown in Wait with the shortest possible ticks). This sending is detected by any node at worst $T_{\max} + \tau_{\max}$ later, which causes a producer in the Ready state to send (a producer in the Wait state has already done so), with the value arriving at the consumer at

[4]The worst-case transmission delay is thus $T_{\max} + \tau_{\max}$.



(a) Property 2: $S^P_{k-1} \prec E^C_k$.



(b) Property 3: $E^C_k \prec S^P_k$.

Figure 6: Explanation of the proofs of properties 2 and 3.

most $\tau_{\max}$ later. Equation (7) guarantees that this happens before the consumer executes.

For property 3, if the $k$th execution of a consumer $E^C_k$ occurs at time $t$ then its $(k-1)$th sending $S^C_{k-1}$ cannot have occurred before $t - pT_{\max}$ (countdown in Wait with the longest possible ticks). The first send by a producer in the $(k-1)$th round $S^p_{k-1}$ cannot occur before $t - pT_{\max} - (T_{\max} + \tau_{\max})$, since any send preempts the consumer in Ready at worst after a delay of $T_{\max} + \tau_{\max}$. Since the smallest delay before the subsequent $k$th send of any producer arrives at the consumer is $pT_{\min} + qT_{\min} + \tau_{\min}$ (countdowns in Wait and Ready with the shortest possible ticks for the first node to publish), equation (8) guarantees that the $k$th execution of the consumer occurs beforehand. □

### *Broadcast Communication.*

The Time-Based protocol does not wait for acknowledgments from all receivers but rather sends a new value as soon as it detects a publication from another node. Controllers thus operate more independently, but broadcast communication is necessary. Otherwise, consider adding a third node $N$ to the scenario in figure 6b such that it communicates with node $P$ but not node $C$. Now, $P$ may be preempted in the Ready state one tick after $E^P_k$ causing it to send a message that arrives at $C$ at $S^P_{k-1} + (p+1)T_{\min} + \tau_{\min}$. Since node $C$ would not be preempted by $N$ but only by $P$, in the worst case $E^C_k$ occurs $(p+1)T_{\max} + \tau_{\max}$ after $S^P_{k-1}$. Property 3 then imposes the impossible condition

$$(p+1)T_{\min} + \tau_{\min} > (p+1)T_{\max} + \tau_{\max}.$$

### *Global Synchronization.*

In fact, properties 2 and 3 imply strictly more than the preservation of the Kahn semantics of an application.

COROLLARY 1. *The Time-Based controller ensures a strict alternation between the* execute *and* send *phases throughout the architecture*

PROOF. Since the parameters $p$ and $q$ are the same for all Time-Based controllers, the following two properties hold:

PROPERTY 4 $(S^C_{k-1} \prec E^P_k)$. *For $k \geq 0$, the $(k-1)$*th *sending of a node is always received at its producers before their respective $k$*th *executions.*

PROPERTY 5 $(E^P_k \prec S^C_k)$. *For $k \geq 0$, the $k$*th *execution of a node always occurs before the $k$*th *send from any of its consumers is received.*

The proofs of these properties resemble those of properties 2 and 3. Since we assume broadcast communication, each node is a potential producer and consumer for all others. Thus the corollary follows directly from properties 2 to 5. □

Compared to the Back-Pressure protocol, the Time-Based protocol forces a global synchronization of the architecture. In fact, running the Back-Pressure protocol under a broadcast assumption also induces such strict alternations since every node must wait for all others to execute before sending a new value (the protocol can be optimized in this case, see appendix E).

### *Performance bounds.*

Optimal performance requires minimal values for $p$ and $q$:[5]

$$p^* = \left\lfloor \frac{2\tau_{\max} + T_{\max}}{T_{\min}} \right\rfloor + 1$$
$$q^* = \left\lfloor \frac{\tau_{\max} - \tau_{\min} + (p+1)T_{\max}}{T_{\min}} - p \right\rfloor + 1.$$

THEOREM 7. *The worst-case throughput of a Time-Based LTTA node is:*

$$\lambda_{\text{TB}} = 1/(p^* + q^*)T_{max}.$$

PROOF. The slowest possible node spends $p^* T_{\max}$ in state WAIT and $q^* T_{\max}$ in state READY. □

## 4.3 Hybrid LTTA

Back-Pressure controllers are architecture-independent and therefore very flexible. However, they violate the principle of 'sender independence' [21, §4.1.1]. If a node crashes and stops sending data, the entire application is stuck forever. In comparison, Time-Based controllers are based on a waiting mechanism tuned with the timing characteristics of the architecture. If a node crashes, other nodes will continue to compute using the last received data. This allows programmers to implement their own fault tolerance policies directly in the application.

The close relationship between the two protocols allows them to be combined within a single system. This technique was introduced in [2], but with the formalization proposed in this paper, the blending of the two LTTA protocols is straightforward and does not require any adaptation. The different components need only be connected together, noting that acknowledgments need not be sent by back-pressure nodes to time-based ones, but that they must be sent by time-based nodes to back-pressure ones.

## 5. CLOCK SYNCHRONIZATION

The LTTA protocols are designed to accommodate the loose timing of node activations in a quasi-periodic architecture. But modern clock synchronization protocols are cost-effective and precise: the Network Time Protocol (NTP) [25]

[5]$\forall x \in \mathbb{R}$, $\lfloor x \rfloor$ denotes the greatest integer $i$ such that $i \leq x$.

and True-Time (TT) [13] provide millisecond accuracies across the Internet, the Precise Time Protocol (PTP) [22] and the Time-Triggered Protocol (TTP) [21, Chapter 8] provide sub-microsecond accuracies at smaller scales. With synchronized clocks, the completely synchronous scheme outlined at the start of section 3 becomes feasible, raising the question: *is there really any need for the LTTA protocols?*

To respond to this question we recall the basics of one of the most efficient clock synchronization schemes in section 5.1, work from well-known principles [21, Chapter 3] to build a globally synchronous system in section 5.2, and finally compare the result with the two LTTA protocols in section 5.3.

## 5.1 Central Master Synchronization

In central master synchronization, a node's local time reference is incremented by the nominal period $T_n$ at every activation. A distinguished node, the *central master*, periodically sends the value of its local time to all other nodes. When a slave node receives this message, it corrects its local time reference according to the sent value and the transmission latency.

For the quasi-periodic architecture, and assuming the central master is directly connected to all other nodes, the maximum difference between local time references immediately after resynchronization depends on the difference between the slowest and the fastest message transmissions between the central master and slaves:

$$\Phi = \tau_{\max} + T_{\max} - \tau_{\min}.$$

Between synchronizations, local time references drift apart. The maximum divergence between any two clocks is $\Gamma = 2\rho R$. The delay between successive resynchronizations $R$ is equal, at best, to the master's activation period. The maximum drift rate $\rho$ is, in our case,

$$\rho = \frac{T_{\max}}{T_n} - 1 = \frac{T_{\max} - T_{\min}}{T_{\max} + T_{\min}}.$$

The maximal precision of the clock synchronization is then

$$\Pi = \Phi + \Gamma.$$

## 5.2 The Global Clock Protocol

A global notion of time can be realized by subsampling the local clock ticks of nodes provided the period of the global clock $T_g$ is greater than the precision of the synchronization, that is, $T_g > \Pi$. This assumption is called the 'reasonableness condition' in [21, Chapter 3, § 3.2.1]. On any given node, the $n$th tick of the global clock occurs as soon as the local reference time is greater than $nT_g$. These particular ticks of the local clocks are called *macroticks*. Because of the reasonableness condition the delay between nodes activations that occur at the same macrotick is less than $\Pi$. Activating nodes on each of their macroticks thus naturally imposes a synchronous execution of the architecture.

Even with synchronization, a value from a faster node may still arrive at a slower one before the latter executes, overwriting the previous value before it can be read. A simple solution, adapted from the synchronous network model of [23, Chapter 2], is to establish separate communication and execution phases. There is, however, no need to execute twice for each activation of the machine. The lock-step execution means that no node can ever execute more than twice between any two activations of another. Communicating through two-place buffers suffices to ensure that messages are never overwritten.

| $T_n$ | $\tau_n$ | $\varepsilon$ | BP | TB | GC |
|---|---|---|---|---|---|
| $10^{-2}$ | $10^{-6}$ | 1% | 2.0 | 4.0 | 3.1 |
| | | 5% | 2.1 | 4.2 | 3.5 |
| | | 15% | 2.3 | 5.7 | 4.5 |
| $10^{-4}$ | $10^{-4}$ | 1% | 4.0 | 6.1 | 3.2 |
| | | 5% | 4.2 | 6.3 | 3.8 |
| | | 15% | 4.6 | 10.3 | 5.4 |
| $10^{-6}$ | $10^{-2}$ | 1% | 2.0 | 2.1 | 1.1 |
| | | 5% | 2.1 | 2.7 | 1.3 |
| | | 15% | 2.3 | 4.6 | 1.9 |

Figure 7: Relative worst case slowdowns for the different protocols: Back-Pressure (BP), Time-Based (TB) and Global Clock (GC), compared to an ideal synchronous execution.

Finally, the transmission delay may prevent a value sent at the $k$th macrotick from arriving before the $(k+1)$th macrotick begins. From the maximum transmission delay, we can calculate the number of macroticks $m$ that a node must wait to sample a new value with certainty:

$$m = \left\lfloor \frac{\tau_{\max}}{T_g} \right\rfloor + 1.$$

This means that the Kahn semantics of an application is preserved if nodes execute one step every $m$ macroticks and communicate through buffers of size two. This gives a worst case throughput of

$$\lambda_{\text{GC}} = 1/mT_g. \tag{9}$$

We call this simple scheme the *Global-Clock protocol.*

## 5.3 Comparative Evaluation

Each of the three protocols entails some overhead in application execution time compared to an ideal scheme where $T_{\min} = T_{\max}$ and $\tau_{\min} = \tau_{\max}$. To give a quantitative impression of their different performance characteristics, we instantiate in figure 7 the worst-case throughputs of the protocols—theorems 4 and 7 and equation (9)—to calculate the slowdown relative to the ideal case for three different architecture classes, from the top: slower nodes/faster communication, comparable nodes and communication, faster nodes/slower communication. In each class, we consider three different jitter values ($\varepsilon$) applied to both the nominal period ($T_n$) and transmission delay ($\tau_n$). The slowdown is the relative application speed for a given architecture and protocol: 1.0 indicates the same speed as an ideal system; 2.0 means twice as slow.

The Global-Clock protocol performs better than both LTTA protocols when the activation period is much less than the transmission delay. In this case, the cost of clock synchronization is negligible and lock-step execution with two-place buffers maximizes application activations. Conversely, when the activation period is much greater than the transmission delay, the Back-Pressure protocol, which does not require node synchronization, performs best.

The Back-Pressure protocol is the least sensitive to jitter as it reacts as soon as fresh values are detected, while the other protocols must wait. The Time-Based protocol is especially sensitive, its performance decreases rapidly as jitter increases.

The slower nodes/faster communication architecture is the closest to the domain we consider (critical control applications). In this case, the Back-Pressure protocol achieves the best worst-case throughput, especially if there is significant jitter, but it does so by introducing inter-node control dependencies. Otherwise the Global-Clock protocol outperforms both others. Note, though, that we consider a simplified and optimistic case; realistic distributed clock synchronization algorithms will have higher overhead. The Time-Based protocol always has the biggest worst-case slowdown, but it is the least intrusive in terms of additional control logic.

## 6. CONCLUSION

In this paper, we presented the *Back-Pressure* and *Time-Based* LTTA protocols in a unified synchronous framework. This gives both a precise description of the implementation of synchronous applications over quasi-periodic architectures, and also permits the direct compilation of protocol controllers together with application functions. The Time-Based protocol that we present is simpler than previously published versions. We show that the Kahn semantics of synchronous applications implemented on quasi-periodic architectures is preserved by both protocols. Finally, we give bounds on the worst-case throughput for the protocols.

The comparison with an optimistic implementation of clock synchronization shows that the LTTA protocols are at least competitive for jittery architectures where the transmission delay is not significant relative to node periods—exactly the class of embedded systems of interest. In addition, the Time-Based protocol is noninvasive and robust: nodes need only listen and wait.

### *Acknowledgements.*

We thank Marc Pouzet and Adrien Guatto for their valuable comments. We also thank Gérard Berry, for the challenging remarks that motivated the comparisons of section 5, and the anonymous reviewers, for their useful suggestions.

## 7. REFERENCES

[1] G. Baudart, A. Benveniste, A. Bouillard, and P. Caspi. A unifying view of Loosely Time-Triggered Architectures. Technical Report RR-8494, INRIA, 2014. Corrected version of [2].

[2] A. Benveniste, A. Bouillard, and P. Caspi. A unifying view of Loosely Time-Triggered Architectures. In *EMSOFT'10*, pages 189–198, 2010.

[3] A. Benveniste, P. Caspi, M. Di Natale, C. Pinello, A. Sangiovanni-Vincentelli, and S. Tripakis. Loosely Time-triggered Architectures based on Communication-by-sampling. In *EMSOFT'07*, pages 231–239, 2007.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1):64–83, 2003.

[5] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for Loosely Time-Triggered Architectures. In *EMSOFT'02*, pages 252–265, 2002.

[6] T. Bourke and M. Pouzet. Zélus: A synchronous language with ODEs. In *HSCC'13*, pages 113–118.

[7] L. P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *ENTCS*, 146(2):61–80, 2006.

[8] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[9] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro*, 22(5):24–35, 2002.

[10] P. Caspi. The quasi-synchronous approach to distributed control systems. Technical Report CMA/009931, VERIMAG, Crysis Project, 2000.

[11] P. Caspi and A. Benveniste. Time-robust discrete control over networked Loosely Time-Triggered Architectures. In *CDC'08*, pages 3595–3600, 2008.

[12] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *PDCS '94*, pages 101–107, 1994.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. In *OSDI'12*, pages 261–264, 2012.

[14] J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *DAC'07*, pages 416–419, 2007.

[15] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.

[16] Esterel Technologies. Scade suite. `http://www.esterel-technologies.com/products/scade-suite/`.

[17] N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. In *EMSOFT'02*, pages 240–251, 2002.

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. IEEE*, 79(9):1305–1320, 1991.

[19] N. Halbwachs and L. Mandel. Simulation and verification of aysnchronous systems by means of a synchronous model. In *ACSD'06*, pages 3–14, 2006.

[20] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP'74*, pages 471–475, 1974.

[21] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer-Verlag, 2nd edition, 2011.

[22] K. Lee and J. C. Eidson. IEEE 1588-standard for a precision clock synchronization protocol for networked measurement and control systems. In *SIcon'02*, pages 98–105, 2002.

[23] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[24] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1–3):61–92, 2001.

[25] D. L. Mills. *Computer Network Time Synchronization: The Network Time Protocol*. Taylor & Francis, 2006.

[26] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale. Implementing synchronous models on Loosely Time Triggered Architectures. *IEEE Trans. on Comp.*, 57(10):1300–1314, 2008.