Loosely Time-Triggered Architectures

Guillaume Baudart, Albert Benveniste, Timothy Bourke

A

Loosely Time-Triggered Architectures: Improvements and Comparisons

GUILLAUME BAUDART, École normale supérieure
ALBERT BENVENISTE, INRIA, Rennes
TIMOTHY BOURKE, INRIA, Paris

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 *nearly 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, a simplified version of the Time-Based protocol and optimizations for systems using broadcast communication. We also compare the LTTA approach with architectures based on clock synchronization.

Categories and Subject Descriptors: C.0 [**General**]: Modeling of Computer Architecture; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems

General Terms: Theory

Additional Key Words and Phrases: Quasi-Periodic Architecture, Loosely Time-Triggered Architecture, Time-Based LTTA, Back-Pressure LTTA

ACM Reference Format:
Guillaume Baudart, Albert Benveniste and Timothy Bourke. 2015. Loosely Time-Triggered Architecture: Improvements and Comparisons. *ACM Trans. Embedd. Comput. Syst.* V, N, Article A (January 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

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) [Kopetz 2011]. Another is to use less constraining protocols as in the Loosely Time-Triggered Architecture (LTTA) [Benveniste et al. 2002; Benveniste et al. 2007; Tripakis et al. 2008; Caspi and Benveniste 2008; Benveniste et al. 2010].

The embedded applications that we consider involve both continuous control and discrete logic. Since the continuous layers are naturally robust to sampling artifacts, continuous components can simply communicate through shared memory without additional synchronization. But the discrete logic for mode changes and similar functionalities is sensitive to such artifacts and requires more careful coordination. The LTTA protocols extend communication by sampling with 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 the undeniable advantages of the latter (like straightforward coordination, determinism, and traceability).

Historically, there are two LTTA protocols: *Back-Pressure* [Tripakis et al. 2008] and *Time-Based* [Caspi and Benveniste 2008]. The Back-Pressure protocol is based on acknowledging the receipt of messages. While 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 [Tripakis et al. 2008; Caspi and Benveniste 2008; Benveniste et al. 2010] 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 [Benveniste et al. 2003] could be used to express the general LTTA framework, its instantiations with specific protocols, and the applications themselves. But we choose Zélus [Bourke and Pouzet 2013] because it also provides a continuous model of time, that allows the direct expression of timing constraints from the underlying network architecture, giving a single, coherent, and precise model. The 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 (the proofs of theorems 5.3 and 5.4 are new), but we give a simpler version of the Time-Based protocol and optimizations for systems using broadcast communication. Finally, modern clock synchronization protocols are now cost-effective and precise [Kopetz 2011; Lee et al. 2005; Mills 2006; Corbett et al. 2012], raising the question: *Is there really any need for the LTTA protocols?* We thus compare the LTTA protocols with approaches based on clock synchronization.

Overview. We start with a brief introduction to Zélus, a synchronous language extended with continuous time. In section 3, we formalize quasi-periodic architectures, model their timing constraints in Zélus, and recall the fundamentals of synchronous applications. Then, in section 4, we present a general framework for modeling controller networks and LTTA protocols. This framework is instantiated in section 5 with the two LTTA protocols, and we present optimizations for networks using broadcast communication in section 6. Finally, in section 7, we compare the protocols to an approach based on clock synchronization.

2. OVERVIEW OF ZÉLUS

Zélus [Bourke and Pouzet 2013] is a first-order dataflow synchronous language extended with Ordinary Differential Equations (ODEs) and hierarchical automata. We present here the basic syntax of its key features.[1]

2.1. Discrete time

The keyword node indicates a discrete stream function. The *clock* of a node refers to the sequence of its successive calls. For instance the following function initializes a countdown o with the value v and decrements it at each step:

```
let node countdown (v) = o where
  rec o = v → (pre o − 1)

val countdown : int →D int
```

[1]More details can be found at http://zelus.di.ens.fr.

where the operators pre(.), the non-initialized unit delay, and . → ., for initialization, are from Lustre [Caspi et al. 1987]. Applying this function to the constant stream of 10s yields the execution:

v	10	10	10	10	10	10	10	10	...
o	10	9	8	7	6	5	4	3	...

The language has *valued signals* built and accessed through the constructs emit and present _ → _. Consider the program:

```
let node positive (i) = s where
  rec present i(v) → do emit s = (v > 0) done
```

val positive : int signal $\xrightarrow{D}$ *bool signal*

Whenever the signal i is emitted with value v, signal s is emitted with value v > 0. A signal is absent if not explicitly emitted. If necessary, a signal can be maintained in a *memory*.

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

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

The keyword init initializes a memory, that is, a variable defined at each activation of the node, and the operator last(.) refers to its previous value.[2] Each time the input signal i is emitted, the memory m is updated with the new received value v. The unit delay (last m) in the definition of o ensures that the output does not depend directly on the input in any given instant. An example of an execution of this node follows.

i		3			5	7		9	...
mi	0	0	0	0	0	0	0	0	...
m	0	3	3	3	5	7	7	9	...
o	0	0	3	3	3	5	7	7	...

Complicated behaviors are often best described using automata whose defining equations at an instant are mode-dependent. An automaton is a collection of states and transitions. Consider the following example.

```
let node elapsed (v) = (s, o) where
  rec c = countdown(v)
  and automaton
      | Wait → do o = false unless (c = 0) then do emit s in Elapsed
      | Elapsed → do o = true done
```

val elapsed : int $\xrightarrow{D}$ *unit signal * bool*

Starting in state Wait, the output o is defined by the equation o = false while the condition (c = 0) is *false*. At the instant that this condition is *true*, that is, when the countdown elapses, signal s is emitted, Elapsed becomes the active state, and the output is thereafter defined by the equation o = true.

[2]The last operator thus behaves like pre for memories initialized with init.

v	3	3	3	3	3	3	...
c	3	2	1	0	−1	−2	...
s				()			...
o	*false*	*false*	*false*	*true*	*true*	*true*	...

2.2. Continuous Time

Zélus combines two models of time: *discrete* and *continuous*. Continuous time functions are introduced by the keyword hybrid. Consider a simple periodic clock that emits a signal every p seconds. Such a clock can be modeled in Zélus using a timer, a simple ODE $\dot{t} = 1$, initialized to the value $-p$, and similarly reinitialized whenever t reaches 0.[3]

```
let hybrid periodic (p) = s where
  rec der t = 1.0 init −.p reset up(last t) → −.p
  and present up(t) → do emit s done

val periodic : float -C-> unit signal
```

The variable t is initialized as described above (init −.p) and increases with slope 1.0 (der t = 1.0). The reinitialization condition is encoded as a *(rising) zero-crossing expression* which a numeric solver monitors to detect and locate significant instants. At zero crossing instants when the last t expression monitored by the up(.) operator passes through zero from a negative value to a positive one, t is reset to the value $-.p$ and the signal s is emitted. In a continuous context, the expression last t refers to the left-limit of signal t. It is needed here to prevent circularity—a so called *causal* or *algebraic* loop—in the definition of t.

Discrete functions can be activated on the presence of signals produced by continuous functions.

```
let hybrid simu () = o where
  rec init o = false
  and c = periodic(2.0)
  and present c() → do (s,o) = elapsed(10) done

val simu : unit -C-> bool
```

A memory o is initialized with the value *false*. Then at each of the events produced by the periodic clock periodic, the new value of o is computed by the discrete function elapsed, otherwise the last computed value is maintained.

3. 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 3.1) and synchronous applications (section 3.3).

3.1. Quasi-Periodic Architectures

Introduced in [Caspi 2000], 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[4] [Halbwachs et al. 1991] 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 [Kopetz 2011], it does not rely on clock

[3] $+.$, $-.$, $*.$, $/.$ denote floating-point operations.
[4] www.esterel-technologies.com/products/scade-suite

synchronization. Such systems arise naturally as soon as two or more microcontrollers running periodic tasks are interconnected. They are common in aerospace, power generation, and railway systems.

Definition 3.1 (Quasi-periodic Architecture). *A* quasi-periodic architecture *is a finite set of nodes $\mathcal{N}$, where every node $n \in \mathcal{N}$ executes nearly periodically, that is, (a) each node starts at $t = 0$, and, (b) the actual time between any two successive activations $T \in \mathbb{R}$ may vary between known bounds during an execution:*

$$0 < T_{min} \leq T \leq T_{max}. \tag{1}$$

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

$$0 < \tau_{min} \leq \tau \leq \tau_{max}. \tag{2}$$

Each is buffered at receivers until replaced by a newer one.

We assume without loss of generality that all nodes start executing at $t = 0$, since initial phase differences between nodes can be modeled by a succession of *mute* activations before the actual start of the system. A quasi-periodic system can also be characterized by its nominal period T_{nom} and maximum jitter ε, where $T_{\text{min}} = T_{\text{nom}} - \varepsilon$ and $T_{\text{max}} = T_{\text{nom}} + \varepsilon$, 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' [Caspi 2000, §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. Any given variable is only 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) [Benveniste et al. 2007].

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 $\tau_{\text{max}} < T_{\text{min}} + \tau_{\text{min}}$, otherwise this assumption only burdens implementations with the technicality of numbering messages and dropping those that arrive out of sequence.

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 3.1 mean, however, that the maximum numbers of consecutive oversamplings and overwritings are functions of the bounds on node periods and transmission delays.

Property 3.2. *Given a pair of nodes executing and communicating according to definition 3.1, the maximum number of consecutive oversamplings and overwritings is*

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

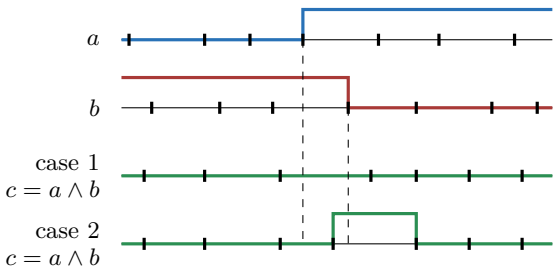


Fig. 1: The effect of sampling on signal combinations.

Proof. Consider a pair of nodes A and B with B receiving messages from A. In the best case, a message sent by A at time t arrives in B's shared memory at $t + \tau_{\min}$. Then if A runs as slowly as possible the next message is sent at $t + T_{\max}$ and arrives in B's shared memory at worst at $t + T_{\max} + \tau_{\max}$. The maximal delay between two successive arrivals is thus

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

At best B is activated every $T_{\min}$. The maximum number of executions n of B is thus:

$$nT_{\min} \leq T_{\max} + \tau_{\max} - \tau_{\min}.$$

Each execution of B that occurs between the two arrivals samples the last received value. The maximum number of oversamplings $n_{os} = n - 1$ is thus given by equation (3). The proof for the number of consecutive overwritings is similar. □

This property implies that data loss can be prevented by activating a consumer more frequently than the corresponding producer, for instance, by introducing *mute* activations of the receiver (at the cost of higher oversampling). Quasi-periodic architectures involving such producer-consumer pairs are studied in [Benveniste et al. 2002].

Quasi-periodic architectures are a natural fit for continuous control applications where the error due to sampling artifacts can be computed and compensated for. 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 [Caspi 2000, §4.2.2][Caspi and Benveniste 2008; Benveniste et al. 2010]. 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 three activations of A before becoming *true* and b is *true* for three 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 changing the frequency of node activations.

3.2. Modeling Quasi-Periodic Architectures

One of the central ideas in the original quasi-synchronous approach is to replace a model with detailed timing behavior by a discrete abstraction [Caspi 2000, §3.2]. Basically, a system is modeled, for example in Lustre, as a composition of discrete programs activated by a *scheduler* program that limits interleaving [Halbwachs and Mandel 2006]. Now, rather than arising as a consequence of the timing constraints of definition 3.1, properties like property 3.2

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' [Caspi 2000, §3.2.1] are replaced by equation (2). In fact, Caspi knew that 'if longer transmission delays are needed, modeling should be more complex' [Caspi 2000, §3.2.1, footnote 2]. The earliest paper on LTTAs [Benveniste et al. 2002] 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 [Caspi and Benveniste 2008] or ad hoc extensions of timed Petri nets [Benveniste et al. 2010]. In contrast, we use Zélus, 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 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 simulation, we express an arbitrary delay by making a random choice.[5]

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

This declares a discrete function named arbitrary with two inputs and defined by a single expression. Then, the model for node clocks is similar to the periodic clock of section 2.2:

```
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 -C-> unit signal
```

The variable t is initialized as described above and increases with slope 1.0. 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 -C-> unit signal
```

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 queues ongoing transmissions which complicates the model without providing any new insights.

3.3. 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

[5] A longer term ambition is to replace this definition to better express the non-determinism of the model.

such a form from a high-level language like Lustre/SCADE, Signal, Esterel [Benveniste et al. 2003], or the discrete part of Simulink[6] does not concern us here.

In the synchronous model, machines are executed in lockstep. 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 3.1 is the subject of section 5.

A Mealy machine m is a tuple $\langle s_{\text{init}}, I, O, F\rangle$, where s_{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 $\mathcal{S}$ is the domain of state values and $\mathcal{V}$ is the domain of variable values. A Mealy machine $m = \langle s_{\text{init}}, I, O, F\rangle$ defines a stream function[7]

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

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 [Maraninchi and Rémond 2001] and distribution over a network [Caspi et al. 1994; Benveniste et al. 2000; Potop-Butucaru et al. 2004] 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 (for $n > 0$, ${}^\bullet x(n) = x(n-1)$). 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* $N = m_1 \,||\, m_2 \,||\, \ldots \,||\, m_p$. The corresponding Mealy machine $N = \langle s_{\text{init}}, I, O, F_N\rangle$ is defined by

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

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 notation used to define F_N describes the shuffling of input, output, and delayed variables.

[6] www.mathworks.com/products/simulink

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

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. Otherwise, it would require synchronization among consumers to avoid non-determinism. 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. Execution order does not matter since no node ever directly depends on the output of another. Thus, at each step, all inputs are consumed simultaneously to immediately produce all outputs. The *Kahn semantics* [Kahn 1974] 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:

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

PROPERTY 3.3. *For Mealy machines, composed as described above, the synchronous semantics and the Kahn semantics are equivalent*[8]

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

PROOF. We write $x :: xs \in \mathcal{V}^\infty$ to represent a stream of values, where $x \in \mathcal{V}$ is the first value of the stream, and $xs \in \mathcal{V}^\infty$ denotes the rest of the stream. Let us first prove for n-tuples of finite or infinite streams *of the same length* that $(\mathcal{V}^n)^\infty \approx (\mathcal{V}^\infty)^n$. We define:

$$\begin{aligned}
&F : (\mathcal{V}^n)^\infty \to (\mathcal{V}^\infty)^n \\
&F(x_1, \ldots, x_n) :: (xs_1, \ldots, xs_n) = (x_1 :: xs_1, \ldots, x_n :: xs_n) \\
&G : (\mathcal{V}^\infty)^n \to (\mathcal{V}^n)^\infty \\
&G(x_1 :: xs_1, \ldots, x_n :: xs_n) = (x_1, \ldots, x_n) :: (xs_1, \ldots, xs_n).
\end{aligned}$$

By construction, streams $x_1 :: xs_1, \ldots, x_n :: xs_n$ all have the same length. Hence, $F \circ G = Id$ and $G \circ F = Id$. This isomorphism can be lifted naturally to functions and we obtain $(\mathcal{V}^I)^\infty \to (\mathcal{V}^O)^\infty \approx (\mathcal{V}^\infty)^I \to (\mathcal{V}^\infty)^O$ for streams of the same length.

Mealy machines always consume and produce streams of the same length since the execution of a Mealy machine consumes all inputs at each step and produces all outputs. The two semantics are thus equivalent. □

The overall idea is to take a synchronous application that has been arranged into a Moore-composition of Mealy machines $N = m_1 \,||\, m_2 \,||\, \ldots \,||\, m_p$, 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.

[8] $A \approx B$ means that A and B are isomorphic.

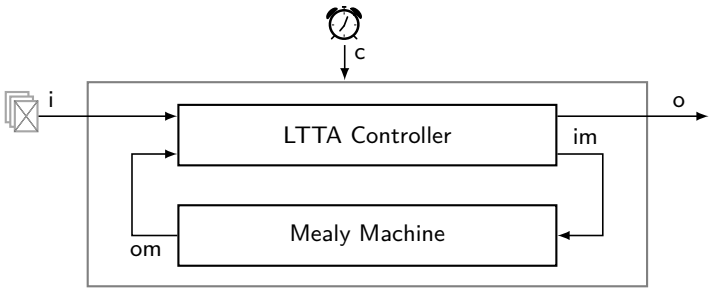


Fig. 2: Schema of an LTTA node: At instants determined by the protocol, the controller samples a list of inputs to triggers the embedded machine, and controls the publication of the output. Symbols ⊠ are implemented by the mem function defined in section 2.1.

4. GENERAL FRAMEWORK

We now consider the implementation of a synchronous application S of p Mealy machines communicating through unit delays on 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 3.1: duplication, loss of data, and unintended signal combinations. We do this by introducing a 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 implementations based on a discrete synchronous model of the architecture. The details of LTTA protocols are presented in section 5.

4.1. From Continuous to Discrete Time

We describe the protocols by adapting a classic approach to architecture modeling using synchronous languages [Halbwachs and Baghdadi 2002]. In doing so, we exploit the ability of the Zélus language to express delays without *a priori* discretization.

The quasi-periodic architecture is modeled by a set of clocks. Signals c1, c2, ... denote the quasi-periodic clocks of the nodes, and dc1, dc2, dc3, ... their delayed versions that model transmission delays (one for each communication channel). The union of all these signals is a global signal g which is emitted on each event. In Zélus, we write:

```
present c1() | c2() | dc1() | dc2() | dc3() | ... → 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.

4.2. 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) [Carloni et al. 2001; Carloni and Sangiovanni-Vincentelli 2002]. 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) done
```

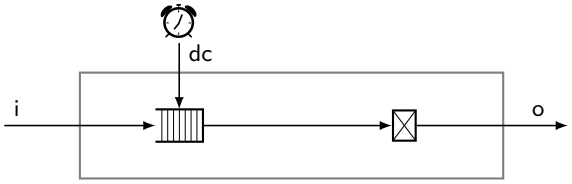


Fig. 3: Schema of communication links modeling delayed transmission between nodes. The striped box represents a FIFO queue.

An LTTA node is modeled in Zélus as:

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

val ltta_node : 'a list --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 (which may be a tuple). The value of om is then 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 (a) when to execute the machine (emission of signal im), and, (b) when to send the resulting outputs (emission of signal o). All the protocols ensure that before sending a new value, the previous one has been read by all consumers. Since nodes execute initially without having to wait for values from other nodes, the LTTA controllers reintroduce the unit delays required for correct distribution.

4.3. 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 3.2). 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 --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 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 the global clock g defined in section 4.1:

```
present g() → do o = link(i) done
```

A link is modeled in Zélus as:

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

val link : unit signal * 'a signal * 'a -D-> 'a
```

When a message is sent on signal i, it goes through the channel and, after the transmission delay modeled by the delayed clock dc, is stored in a memory. New messages overwrite previous memory values. The memory contents are output by the link. Note that the memory mem imposes a unit delay between the input i and the output o thus forbidding instantaneous transmission (section 2.1). Since we assume that node computations do not depend on the initial values of delayed outputs (section 3.3), we can initialize the memories of LTTA links with an arbitrary value mi.

Fresh values. The LTTA controllers must detect when a fresh write is received in an attached shared memory even when the same value is sent successively. 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 -D-> 'a msg signal
```

The value of the boolean variable flag is paired with each new value received on signal i. Its 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, mi) = o where
  rec s = channel(dc, i)
  and o = mem(alternate(s), mi)

val ltta_link : unit signal * 'a signal * 'a msg -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, st) = o where
  rec init m = st
  and present r(_) → do m = i.alt done
  and o = (i.alt <> last m)

val fresh : 'a msg * 'b signal * bool -D-> bool
```

Variable m stores the alternating bit associated with the last read value. It 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, when i.alt <> last m. The boolean flag st states whether or not the initial value is considered as fresh.

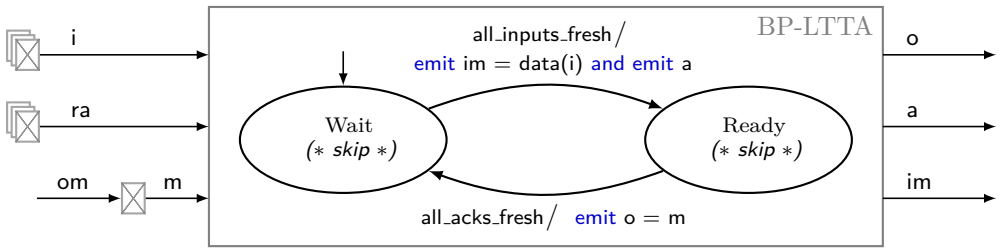


Fig. 4: The Back-Pressure LTTA controller. The additional inputs ra are acknowledgments from consumers. The additional output a is for acknowledging producers.

5. THE LTTA PROTOCOLS

We now present the LTTA protocols. There are two historical proposals, one based on back-pressure (section 5.1), and another based on time (section 5.2); and two optimizations for networks using broadcast communication (section 6).

5.1. Back-Pressure LTTA

The Back-Pressure protocol [Tripakis et al. 2008] is inspired by *elastic circuits* [Cortadella and Kishinevsky 2007; Cortadella et al. 2006] where a consumer node must acknowledge each value read by writing to a *back pressure* link [Carloni 2006] 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 [Tripakis et al. 2008] and included in later Petri net formalizations [Benveniste et al. 2010; Baudart et al. 2014].

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, om, mi) = (o, a, im) where
  rec m = mem(om, mi)
  and 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, true)
  and all_acks_fresh = forall_fresh(ra, o, false)

val bp_controller :
'a msg list * 'b msg list * 'c signal * 'c -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(.) accesses the data field of the msg structure), stores the result in a local memory m, sends an acknowledgment to the producer, and transitions immediately to Ready. The controller skips in Ready until acknowledgments 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.

The freshness of the inputs since the last execution of the machine is tested by a conjunction of fresh nodes (forall_fresh(i, im, true)). The controller also tests whether fresh acknowledgments have been received from all consumers since the last emission of the output signal o.[9]

Remark 5.1. The composition of a Back-Pressure controller and a Mealy machine to form an LTTA node is well defined. Indeed, the dependency graph of the controller is:

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

Since the communication with the embedded machine adds the dependency $\mathsf{om} \leftarrow \mathsf{im}$, the composition of the two machines is free of cycles and therefore well defined.

Preservation of Semantics. This result was first proved in [Tripakis et al. 2008] for networks of nodes communicating through buffers of arbitrary size. Another proof is given in [Benveniste et al. 2010; Baudart et al. 2014] based on the relation with elastic circuits. We give here a new straightforward proof based on the following *liveness* property.

PROPERTY 5.2. *Let $t(E_k^N)$ be the date of the kth execution of the embedded machine of a node N. For $k > 0$, and for any node N, we have:*

$$t(E_k^N) \leq 2(\tau_{max} + T_{max})(k-1).$$

PROOF. This property is shown by induction on k.

Initialization. Since all nodes start at $t = 0$ and since they can execute immediately without having received values from other nodes, we have for all nodes N, $t(E_1^N) = 0$.

Induction. Assume the property holds up to and including k. At worst, the last node executes and sends an acknowledgment at $t = 2(\tau_{\max}+T_{\max})(k-1)$. The last acknowledgment is thus received at worst $\tau_{\max}$ later, just after a tick of a receiver's clock. Therefore the receiver does not detect the message until $t + \tau_{\max} + T_{\max}$.[10] The latest kth publication then occurs at $t + \tau_{\max} + T_{\max}$. Symmetrically this publication is detected at worst $\tau_{\max} + T_{\max}$ later. Hence the $(k+1)$th execution occurs at $t + 2(\tau_{\max} + T_{\max})$, that is, at $2(\tau_{\max} + T_{\max})k$. □

Consequently, in the absence of crashes, nodes never block, which is enough to ensure the preservation of semantics.

THEOREM 5.3 ([TRIPAKIS ET AL. 2008; BENVENISTE ET AL. 2010]). *Implementing a synchronous application S over a quasi-periodic architecture (definition 3.1) with Back-Pressure controllers preserves the Kahn semantics of the application:*

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

PROOF. Back-Pressure controllers ensure that nodes always sample fresh values from the memories (guard all_inputs_fresh) and never overwrite a value that has not yet been read (guard all_acks_fresh). Since property 5.2 ensures that nodes will always execute another step, the Kahn semantics of the application is preserved. □

Performance Bounds. Property 5.2 also allows the analysis the worst-case performance of Back-Pressure LTTA nodes.

THEOREM 5.4 ([TRIPAKIS ET AL. 2008; BENVENISTE ET AL. 2010]). *The worst case throughput of a Back-Pressure LTTA node is*

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

[9]Initially there are no fresh acknowledgements since controllers start in the Wait state.
[10]The worst-case transmission delay on a quasi-periodic architecture is $T_{\max} + \tau_{\max}$.

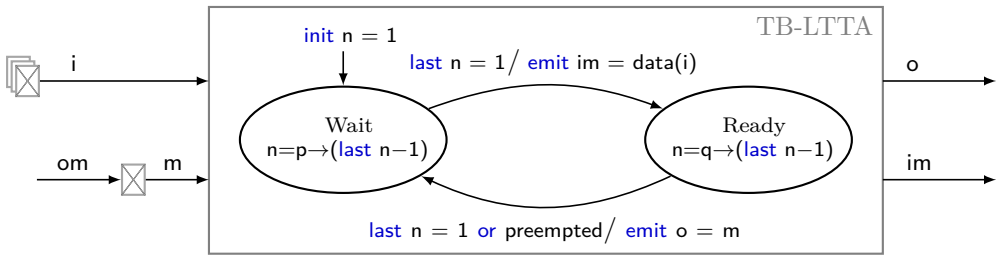


Fig. 5: The Time-Based LTTA controller. A counter n is decremented in each state initialized with value p in state WAIT and q in state READY; preempted indicates that a fresh value was received on some input.

PROOF. This result follows from property 5.2. In the worst case, the delay between two successive executions of a node is $2(T_{\max} + \tau_{\max})$. □

5.2. Time-Based LTTA

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

A first version of the Time-Based LTTA protocol was introduced in [Caspi 2000]. The protocol was formalized as a Mealy machine with five states in [Caspi and Benveniste 2008] and a simplified version was modeled with Petri nets in [Benveniste et al. 2010; Baudart et al. 2014]. We propose an even simpler version that can be expressed as a two-states automaton, formalize it in Zélus, and prove its correctness.

Unlike the Back-Pressure protocol, the Time-Based protocol requires *broadcast communication* and acknowledgment values are not sent when inputs are sampled.

ASSUMPTION 1 (BROADCAST COMMUNICATION). *All variable updates must be visible at all nodes and each node must update at least one variable.*

The controller for the Time-Based protocol is shown in figure 5, for parameters p and q:

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

  and preempted = exists_fresh(i, im, true)

val tb_controller : 'a msg list ∗ 'b signal ∗ 'b -D-> 'b 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, stores the result in the local memory m, and enters Ready. In Ready, the equation n = q → (last 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 passes directly into the Wait state, resets the counter to p, and *sends* the previously computed outputs from the memory m 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 accommodate the unsynchronized activations of other nodes and message transmission delays, but accelerate when they detect a message from other nodes.

Remark 5.5. The composition of a Time-Based controller and a Mealy machine to form an LTTA node is always well defined. The proof is similar to that of *remark* 5.1. The dependency graph of a node is:

$$\mathsf{n} \leftarrow \mathsf{i} \quad \mathsf{o} \leftarrow \mathsf{i} \quad \mathsf{o} \leftarrow \mathsf{m} \quad \mathsf{om} \leftarrow \mathsf{im} \quad \mathsf{im} \leftarrow \mathsf{i}.$$

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. Similar results can be found in [Caspi and Benveniste 2008; Benveniste et al. 2010; Baudart et al. 2014] for previous versions of the protocol.

THEOREM 5.6. *The Kahn semantics of a synchronous application S implemented on a quasi-periodic architecture (definition 3.1) with broadcast communication (assumption 1) 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 kth 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 5.7 ($S^P_{k-1} \prec E^C_k$). *For $k > 0$, the $(k-1)$*th *sending of a producer is received at its consumers before their respective k*th *executions.*

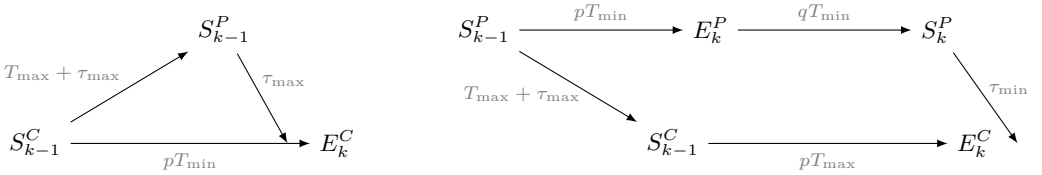
PROPERTY 5.8 ($E^C_k \prec S^P_k$). *For $k > 0$, the k*th *execution of a consumer occurs before the k*th *sending from any of its producers is received.*

The properties are shown by induction on k.

Initialization. Nodes start at $t = 0$ and execute immediately (E^C_1) without having to receive values from other nodes. The slowest possible consumer first executes at $pT_{\max}$. On the other hand, the smallest delay before the first 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). From equations (7) and (8) we then have

$$(p+q)T_{\min} + \tau_{\min} > \tau_{\max} + (p+1)T_{\max} > pT_{\max},$$

which guarantees that the consumer executes before the reception of the new value.



(a) Property 5.7: $S^P_{k-1} \prec E^C_k$. (b) Property 5.8: $E^C_k \prec S^P_k$.

Fig. 6: Explanation of the proofs of properties 5.7 and 5.8.

Induction. Assume that the properties hold up to and including $k-1$. The proofs proceed by considering the worst-case scenarios illustrated in Figure 6.

For property 5.7, if the kth 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 most $\tau_{\max}$ later. Equation (7) guarantees that this happens before the consumer executes. If node C was not the first to send the $(k-1)$th value, S^P_{k-1} would have occured even earlier.

For property 5.8, if the kth 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 kth 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 kth 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 the scenario of figure 7 obtained by 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 5.8 would then require the impossible condition

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

Global Synchronization. In fact, properties 5.7 and 5.8 imply strictly more than the preservation of the Kahn semantics of an application.

Corollary 5.9. *The Time-Based controller ensures a strict alternation between* execute *and* send *phases throughout the architecture.*

Proof. Since the Time-Based protocol requires broadcast communication, each node is a producer and consumer for all others. Therefore, properties 5.7 and 5.8 impose a strict alternation between *execute* and *send* phases. □

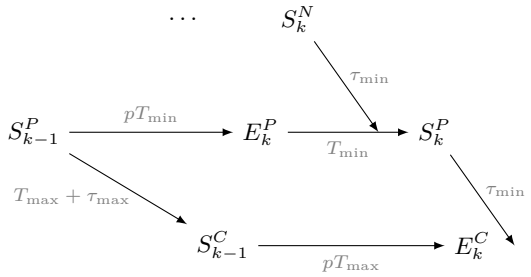


Fig. 7: Behavior of the Time-Based protocol without broadcast communication. Node N preempts node P but not node C. Then node P preempts node C.

Performance bounds. Optimal performance requires minimal values for p and q:[11]

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

THEOREM 5.10. *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 WAIT and $q^*T_{\max}$ in READY. □

Note that this case only occurs if all nodes are perfectly synchronous and run as slowly as possible. Otherwise, slow nodes would be preempted by the fastest one, thus improving the overall throughput. To give a rough comparison with theorem 5.4, remark that we have $p, q \geq 2$ thus, in any case $\lambda_{\text{TB}} \leq 1/4T_{\max}$. A more detailed comparison can be found in section 7.3.

6. OPTIMIZATIONS

Compared to the Back-Pressure protocol, the Time-Based protocol forces a global synchronization of the architecture. But running the Back-Pressure protocol under the same broadcast assumption (assumption 1) also induces such strict alternations since every node must wait for all others to execute before sending a new value. However, when all nodes communicate by broadcast, there are simpler and more efficient alternatives. We propose here two optimizations for these particular networks.

6.1. Round-Based LTTA

The idea of the Round-Based controller is to force a node to wait for messages from all other nodes before computing and sending a new value. Nodes together perform rounds of execution. Unfortunately, at the start of a round, a value sent from a faster node may be received at a slower one and overwrite the last received value before the latter executes. A simple solution, based on the synchronous network model [Lynch 1996, Chapter 2], is to introduce separate communication and execution phases. In this case, we could simply execute each application every two rounds. But since lock-step execution ensures that no node can execute more than twice between two activations of any other, it is enough to communicate via buffers of size two. This ensures that messages are never overwritten even if nodes execute the application

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

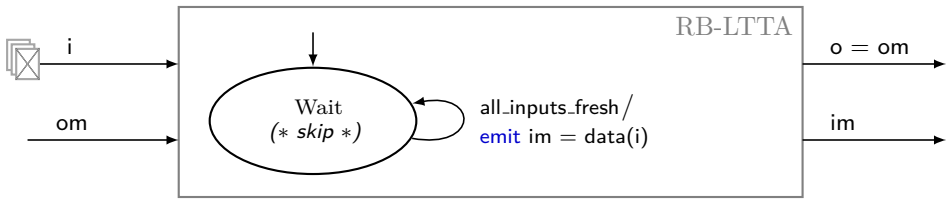


Fig. 8: The Round-Based controller. Acknowledgment are no more required. When all inputs are detected, the controller triggers the embedded machine and directly sends the output om to other nodes.

and directly send the output at every activation. Acknowledgments are no longer required. The Zélus code of the controller shown in figure 8 is:

```
let node rb_controller (i, om) = (o, im) where
  rec automaton
      | Wait →
          do (* skip *)
          unless all_inputs_fresh then
            do emit im = data(i) in Wait

  and all_inputs_fresh = forall_fresh(i, im, true)
  and o = om

val rb_controller : 'a msg list * 'b -D-> 'b * 'a list signal
```

The forall_fresh now indicates that all input buffers contain at least one value.

Compared to the Back-Pressure and Time-Based protocols, a local memory is not required to store the result of the embedded Mealy machine since machine's output is immediately sent to other nodes.

Remark 6.1. The composition of a Round-Based controller and a Mealy machine to form an LTTA node is always well defined. The proof is again similar to that of *remark* 5.1. The dependency graph of a node is:

$$\mathsf{o} \leftarrow \mathsf{om} \quad \mathsf{om} \leftarrow \mathsf{im} \quad \mathsf{im} \leftarrow \mathsf{i}.$$

It has no cyclic dependencies.

Preservation of the semantics. For systems using broadcast communication (assumption 1), Round-Based controllers induce a synchronous execution throughout the entire system thus ensuring the preservation of the Kahn semantics. All nodes execute at approximately the same time.

Performance bounds. Compared to nodes controlled by the Back-Pressure protocol, Round-Based nodes can be twice as fast since they immediately send the output of the embedded machine at each step.

THEOREM 6.2. *The worst case throughput of a Round-Based LTTA node is:*

$$\lambda_{\mathrm{RB}} = 1/(T_{max} + \tau_{max}).$$

PROOF. Suppose that the last execution of the $(k-1)$th round occurs at time t. In the worst case, a node detects this last publication and sends its new message at $t + \tau_{\max} + T_{\max}$. The last execution of the kth round thus occurs $\tau_{\max} + T_{\max}$ after the last execution of the previous round. □

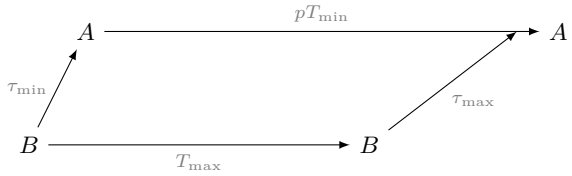


Fig. 9: Explanation of the proof of property 6.3.

6.2. Timed Round-Based LTTA

Like the Back-Pressure protocol, the Round-Based protocol uses blocking communication. If a node crashes, the entire application stops. To avoid such problems, a classic idea is to add timeouts [Attiya et al. 1994] and to run a crash detector together with the Round-Based controller on each node. When a controller executes a step of the application, it knows which other nodes are still functioning, since it has received messages from them, and which have crashed. It can continue to compute using the values last received from crashed nodes.

At each activation, nodes broadcast a *heartbeat* message to signal that they are still active. Every node A maintains a counter initialized to a value p for each other node. The counter corresponding to a node B is reset to its initial value whenever a *heartbeat* message is received from B. The following property ensures that when the counter reaches zero, node A can conclude that B has crashed.

Property 6.3 ([Attiya et al. 1994]). *For all nodes A, the counter associated to another node B can only reach zero if B crashed, provided that:*

$$p > \frac{\tau_{max} - \tau_{min} + T_{max}}{T_{min}} \tag{9}$$

Proof. The proof involves considering the worst case scenario illustrated in figure 9. Each time a node B executes, it sends a *heartbeat* message to A. The maximum difference between the times of two consecutive sends is $T_{\max}$. In the worst case, A receives the first message after the shortest possible delay $\tau_{\min}$ and the second after the longest possible delay $\tau_{\max}$. If A runs as fast as possible the counter reaches zero $pT_{\min}$ after the reception of the first message. Hence the condition $\tau_{\min} + pT_{\min} > \tau_{\max} + T_{\max}$ suffices to ensure that the counter only reaches zero if node B has crashed. □

The Zélus code for the timeout mechanism is:

```
let node timeout (i_live) = o where
  rec reset n = p → pre n − 1 every i_live
  and o = (n = 0)

val timeout : bool D→ bool
```

There is one additional boolean input i_live for each node. It indicates if a *heartbeat* message has been received since the last activation.

A node executes a step of the application if for every other node it has either received a fresh message or detected a crash. In our model, we need only replace the implementation of fresh(i, r, st) (section 4.3) with:

```
let node timed_fresh (i, i_live, r, st) =
  fresh(i, r, st) or timeout(i_live)

val timed_fresh : 'a msg ∗ bool ∗ 'b signal ∗ bool D→ bool
```

Performance bounds. In the absence of crashes the timeout mechanism has no influence on the behavior of nodes (property 6.3) and the Timed Round-Based protocol coincides with the Round-Based one. Otherwise the minimal value for the initial value p is:

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

When one or more nodes crash, active nodes wait at worst $p^*T_{\max}$ before detecting the problem and only then execute a step of the application and send the corresponding message. The delay between two successive rounds is thus bounded by $p^*T_{\max} + \tau_{\max}$.

Since every node broadcasts a message at every step, the timeout mechanism has a high message complexity. An alternative is to send a *heartbeat* message only once every k steps and to adjust the initial value of the counters appropriately. The worst case delay between two successive rounds increases accordingly.

7. 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) [Mills 2006] and True-Time (TT) [Corbett et al. 2012] provide millisecond accuracies across the Internet, the Precise Time Protocol (PTP) [Lee et al. 2005] and the Time-Triggered Protocol (TTP) [Kopetz 2011, Chapter 8] provide sub-microsecond accuracies at smaller scales. With synchronized clocks, the completely synchronous scheme outlined at the start of section 4 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 7.1. Then we work from well-known principles [Kopetz 2011, Chapter 3] to build a globally synchronous system in section 7.2. Finally we compare the result with the two LTTA protocols and their round-based counterparts in section 7.3.

7.1. Central Master Synchronization

In central master synchronization, a node's local time reference is incremented by the nominal period T_{nom} 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. This synchronization scheme is illustrated in figure 10.

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:

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

The delay between successive resynchronizations R is equal, at best, to the master's activation period. Between synchronizations, a node clock may drift from the master clock. The maximum drift rate ρ is, in our case,

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

The optimal precision of clock synchronization is then the maximal accumulated divergence between two node clocks during the resynchronization interval, that is,

$$\Pi = \Phi + 2\rho R.$$

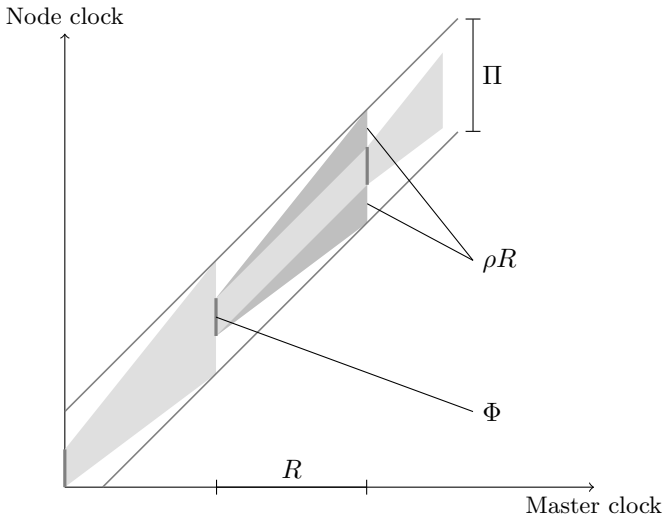


Fig. 10: [Kopetz 2011, Figure 3.10] Central Master Synchronization: a node's clock stays within the entire shaded area. R denotes the resynchronization interval, Φ the offset after resynchronization, ρ the drift rate between two clocks, and Π the precision of the protocol.

7.2. 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 [Kopetz 2011, Chapter 3, §3.2.1]. On any given node, the nth 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*. Under the reasonableness condition the delay between nodes activations that occur at the same macrotick is less than Π. Activating nodes on each of their macroticks thus naturally imposes a synchronous execution of the architecture. Then, as for the round-based protocols, communication through two-place buffers suffices to ensure that messages are never incorrectly overwritten.

Finally, the transmission delay may prevent a value sent at the kth 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

$$\lambda_{\text{GC}} = 1/mT_g. \tag{10}$$

We refer to this simple scheme as the *Global-Clock protocol*.

7.3. Comparative Evaluation

Performances. Each of the 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 table I the worst-

T_{nom}	τ_{nom}	ε	BP	TB	RB/TRB	GC
10^{-2}	10^{-6}	1%	2.0	4.0	1.0	3.1
		5%	2.1	4.2	1.0	3.5
		15%	2.3	5.7	1.1	4.5
10^{-4}	10^{-4}	1%	4.0	6.1	2.0	3.2
		5%	4.2	6.3	2.1	3.8
		15%	4.6	10.3	2.3	5.4
10^{-6}	10^{-2}	1%	2.0	2.1	1.0	1.1
		5%	2.1	2.7	1.0	1.3
		15%	2.3	4.6	1.1	1.9

Table I: Relative worst case slowdowns for the different protocols: Back-Pressure (BP) and Time-Based (TB); the optimizations Round-Based (RB) and Timed Round-Based (TRB); and Global Clock (GC), compared to an ideal synchronous execution.

case throughputs of the protocols—theorems 5.4, 5.10 and 6.2 and equation (10)— and calculate the slowdown relative to the ideal case for three different classes of architecture, from the top: slower nodes/faster communication, comparable nodes and communication, faster nodes/slower communication. In each class, we consider three different jitter values (ε) applied to both the nominal period (T_{nom}) and transmission delay (τ_{nom}). 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 shows the best performances 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. For the same reason, protocols optimized for systems using broadcast communication outperform both historical LTTA protocols and the Global Clock protocol, which still requires a little overhead for synchronization (slowdown factor between 1.1 and 1.9). Conversely, when the activation period is much greater than the transmission delay, the Time-Based protocol, which waits for the slowest nodes has the worst performances. Also, in that case, the overhead due to clock synchronization becomes significant and protocols that do not require this synchronization perform best.

The Time-Based protocol is especially sensitive to jitter, its performance decreases rapidly as jitter increases. Rather than waiting for messages from all other nodes, the Time-Based protocol only needs the very first message received in a round and then waits long enough to be sure that all other messages have been received. It is thus more pessimistic than the Round-Based protocol which reacts a soon as all inputs are detected.

In all cases, Round-Based protocols achieve the best worst-case throughput, especially if there is significant jitter, and the two historical protocols (BP and TB) show comparable or worse performances than those of the Global-Clock protocol. 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.

Fault Tolerance. The Back-Pressure and Round-Based protocols rely on blocking communication. If a node crashes, the entire system stops. Therefore fault tolerance mechanisms must be implemented in the middleware (for instance, resurrection mechanisms). On the other hand, the Time-Based, Timed Round-Based, and Global Clock protocols use timing mechanisms. If a node crashes, active nodes continue computing using the values last sent by the crashed node. This behavior allows fault tolerance mechanisms to be implemented in the application. We only consider fail-stop crashes. Fault-tolerance in the general case

with omission or byzantine failures is a complex problem that requires more sophisticated protocols (with voters, self-checking, agreement protocols, clique avoidance, and node reintegration) [Kopetz and Bauer 2003]. The LTTA protocols aim only to provide a lighter alternative for less demanding systems.

8. CONCLUSION

In this paper, we presented the *Back-Pressure* and *Time-Based* LTTA protocols and optimizations of these protocols for systems using broadcast communication 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.[12] We show that the Kahn semantics of synchronous applications implemented on quasi-periodic architectures is preserved by all protocols. Finally, we give bounds on the worst-case throughputs of the protocols.

The comparison with an optimistic implementation of clock synchronization shows that the LTTA protocols and their optimizations 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, LTTA protocols are simple to implement: nodes need only listen and wait and can thus be implemented as one- or two-state automata.

Acknowledgments. We thank Marc Pouzet, Adrien Guatto, and Thibault Rieutord for their valuable comments. We also thank Gérard Berry for the challenging remarks that motivated the comparisons of section 7, and the anonymous reviewers, for their useful suggestions.

REFERENCES

Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1994. Bounds on the time to reach agreement in the presence of timing uncertainty. *JACM* 41, 1 (1994), 122–152.

Guillaume Baudart, Albert Benveniste, Anne Bouillard, and Paul Caspi. 2014. *A Unifying View of Loosely Time-Triggered Architectures*. Technical Report RR-8494. INRIA. Corrected version of [Benveniste et al. 2010].

Albert Benveniste, Anne Bouillard, and Paul Caspi. 2010. A unifying view of loosely time-triggered architectures. In *EMSOFT'10*. USA, 189–198.

Albert Benveniste, Benoît Caillaud, and Paul Le Guernic. 2000. Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation. *Information and Computation* 163, 1 (2000), 125–171.

Albert Benveniste, Paul Caspi, Marco Di Natale, Claudio Pinello, Alberto L. Sangiovanni-Vincentelli, and Stavros Tripakis. 2007. Loosely Time-Triggered Architectures based on Communication-by-Sampling. In *EMSOFT'07*. Austria, 231–239.

Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. of the IEEE* 91, 1 (2003), 64–83.

Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. 2002. A Protocol for Loosely Time-Triggered Architectures. In *EMSOFT'02*. France, 252–265.

Timothy Bourke and Marc Pouzet. 2013. Zélus: A Synchronous Language with ODEs. In *HSCC'13*. USA, 113–118.

Luca P. Carloni. 2006. The Role of Back-Pressure in Implementing Latency-Insensitive Systems. *Elecronic Notes in Theoretical Computer Science* 146, 2 (2006), 61–80.

Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (2001), 1059–1076.

Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. 2002. Coping with latency in SoC design. *IEEE Micro* 22, 5 (2002), 24–35.

Paul Caspi. 2000. *The Quasi-Synchronous Approach to Distributed Control Systems*. Technical Report CMA/009931. VERIMAG, Crysis Project. "The Cooking Book".

[12]Source code is available at http://www.baudart.eu/tecs2016.

Paul Caspi and Albert Benveniste. 2008. Time-robust discrete control over networked loosely time-triggered architectures. In *CDC'08*. Mexico, 3595–3600.

Paul Caspi, Alain Girault, and Daniel Pilaud. 1994. Distributing Reactive Systems. In *PDCS'94*. 101–107.

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *POPL'87*. 178–188.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. 2012. Spanner: Google's globally distributed database. In *OSDI'12*. USA, 261–264.

Jordi Cortadella and Michael Kishinevsky. 2007. Synchronous Elastic Circuits with Early Evaluation and Token Counterflow. In *DAC'07*. USA, 416–419.

Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Christos P. Sotiriou. 2006. Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 25, 10 (2006), 1904–1921.

Nicolas Halbwachs and Siwar Baghdadi. 2002. Synchronous modelling of asynchronous systems. In *EMSOFT'02*. France, 240–251.

Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous dataflow programming language Lustre. *Proc. of the IEEE* 79, 9 (1991), 1305–1320.

Nicolas Halbwachs and Louis Mandel. 2006. Simulation and Verification of Aysnchronous Systems by means of a Synchronous Model. In *ACSD'06*. Finland, 3–14.

Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *IFIP'74*. Sweden, 471–475.

Hermann Kopetz. 2011. *Real-time systems: design principles for distributed embedded applications*. Springer-Verlag.

Hermann Kopetz and Günther Bauer. 2003. The time-triggered architecture. *Proc. of the IEEE* 91, 1 (2003), 112–126.

Kang Lee, John C. Eidson, Hans Weibel, and Dirk Mohl. 2005. *IEEE 1588-Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Technical Report.

Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Florence Maraninchi and Yann Rémond. 2001. Argos: an automaton-based synchronous language. *Computer Languages* 27, 1–3 (2001), 61–92.

David L. Mills. 2006. *Computer Network Time Synchronization, The Network Time Protocol*. Taylor & Francis Group.

Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. 2004. Concurrency in Synchronous Systems. In *ACSD'04*. 67–78.

Stavros Tripakis, Claudio Pinello, Albert Benveniste, Alberto L. Sangiovanni-Vincent, Paul Caspi, and Marco Di Natale. 2008. Implementing synchronous models on Loosely Time-Triggered Architectures. *IEEE Trans. on Computers* 57, 10 (2008), 1300–1314.