Compositional design of isochronous systems

Jean-Pierre Talpin[a], Julien Ouy[a], Thierry Gautier[a], Loïc Besnard[a], Paul Le Guernic[a]

[a] *INRIA, Unité de Recherche Rennes-Bretagne-Atlantique and CNRS, UMR 6074*
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract

The synchronous modeling paradigm provides strong correctness guarantees for embedded system design while requiring minimal environmental assumptions. In most related frameworks, global execution correctness is achieved by ensuring the insensitivity of (logical) time in the program from (real) time in the environment. This property, called endochrony or patience, can be statically checked, making it fast to ensure design correctness. Unfortunately, it is not preserved by composition, which makes it difficult to exploit with component-based design concepts in mind. Compositionality can be achieved by weakening this objective, but at the cost of an exhaustive state-space exploration. This raises a tradeoff between performance and precision. Our aim is to balance it by proposing a formal design methodology that adheres to a weakened global design objective: the non-blocking composition of weakly endochronous processes, while preserving local design objectives for synchronous modules. This yields an effective and cost-efficient approach to compositional synchronous modeling.

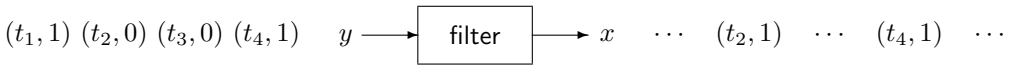
1. Introduction

The synchronous paradigm [5] provides strong guarantees about the correct execution of embedded software while requiring minimal assumptions on their execution environment. In most synchronous formalisms, this is achieved by locally verifying that computation (in the system, in the program) is insensitive to communication delays (from the environment, from the network), i.e., that a process or synchronous module is "patient" (in latency-insensitive design [7]) or "endochronous" (literally, "time is defined from inside" [13]) or a Kahn process (the output flow is a monotonic function of the input stream, the output clock a sample of the input clock, as in Lustre [8]).

Example. For instance, consider a filtering process that emits an event along the output signal x every time the value of its input signal y changes. Each event is denoted by a time tag $t_{1..4}$ and a value $0, 1$. We notice that all output tags $t_{2,4}$ are related to the input tags $t_{1..4}$. This means that process filter is not only deterministic: its output value is a function of its input value at all times; but it also maintains an invariant timing relation

Email addresses: Jean-Pierre.Talpin@inria.fr (Jean-Pierre Talpin), Julien.Ouy@inria.fr (Julien Ouy), Thierry.Gautier@inria.fr (Thierry Gautier), Loic.Besnard@irisa.fr (Loïc Besnard), Paul.LeGuernic@inria.fr (Paul Le Guernic)

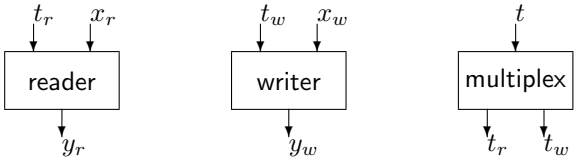
between its input and its output: each output tag is a function of the input tags and values at all times.



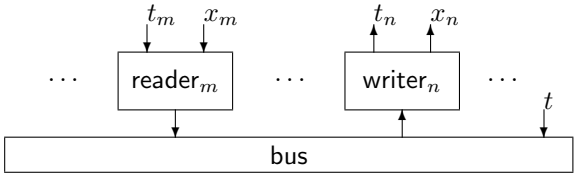
Hence, we can say that process filter is patient (its local timing behavior is independent of external timing constraints) or endochronous (it makes an internal sense of time) or a Kahn process (the output time and value are a function of the input time and value)

In data-flow synchronous formalisms [8, 13], design is usually driven by this very same safety objective. Each individual process must guarantee that its internal synchronization of computations and communications is independent of possible external latency. However, endochrony or patience is not a compositional property: it is not preserved by synchronous composition.

Example. Suppose we wish to build a system using this objective of endochrony in mind. We start by considering elementary blocks, readers (left) and writers (middle). We assume that they are individually endochronous: each output signal y is a function of an input signal x timed by an input clock t. Next, suppose we wish to compose a reader and a writer. The program we obtain is no longer endochronous: the output of the system (y_r or y_w) is no longer timely related to an individual clock (t_r or t_w) unless one adds a multiplexer block (right) to define (and synchronize) the reader and writer clocks $t_{r,w}$ by a function of the master clock t.



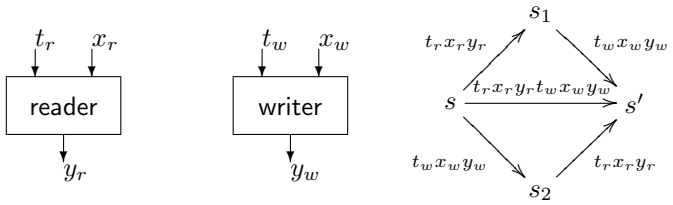
An implementation challenge. Now, consider a more realistic implementation scenario in which we need to build the simulator for an embedded architecture consisting of thousands of individually endochronous reader and writer processes communicating via a loosely time-triggered bus. This is typically the case in modern embedded avionics or automotive architectures.



We see that endochrony is not a reasonable objective to design such as system: we cannot afford to manually build a global controller to synchronize all individual components in the system. Design would take a lot of time to manage all local timing constraints (if at all doable), the implementation would perform slowly (all blocks would have to synchronize on a global tick), and the design would have to be updated for every new block added to or changed in the system. The main concern for such a large system is compositionality.

In [20], it is shown that compositionality can be achieved by weakening the objective of endochrony: a weakly endochronous system is simply a deterministic system that can perform independent computations and communications (here, possibly, each individual block) in any order as long as, of course, this does not alter its global state. Such a system satisfies the so called diamond property of concurrency theory. In [20], it is further shown that the non-blocking composition of weakly endochronous processes is isochronous. This means that the synchronous and asynchronous compositions of weakly endochronous processes accept the same behaviors. As a result, weak endochrony appears to be a much more suitable criterion for our design problem: it is compositional and imposes less design constraints on individual modules.

Example. As an example weakly endochronous interaction, consider the interleaved behavior of two deterministic processes, a reader and a writer, depicted using the automaton on the right. From any reachable state s, choosing to perform $t_r x_r$ and y_r first (resp. $t_w x_w y_w$) should not alter the possibility to perform $t_w x_w$ and y_w from s_1 (resp. $t_r x_r y_r$) or to perform both actions simultaneously (i.e. during the same transition).



We observe that checking that a system is weakly endochronous requires an exhaustive exploration of its state-space to guarantee that its behavior is independent from the order of inbound communications. This raises an analytic tradeoff between performance (incurred by state-space exploration) and flexibility (gained from compositionality). We balance this trade-off by proposing a formal design methodology that weakens the global design objective (non-blocking composition) and preserves design objectives secured locally (by accepting patient components). This yields a less general (more abstract) yet cost-efficient approach to compositional modeling that is able to encompass most of the practical engineering situations. It is particularly aimed at efficiently reusing most of the existing program analysis and compilation algorithms of Signal. To implement the present design methodology, we have designed a simple scheduler synthesis and code generation scheme, presented in [19].

Plan. The article starts in Section 2 with an introduction to Signal and its polychronous model of computation. Section 3 defines the necessary analysis framework and Section 4 present our contributed formal properties and methodology. We review related works in Section 6 and conclude.

2. An introduction to polychrony

In Signal, a process (written P or Q) consists of the synchronous composition (noted $P \,|\, Q$) of equations on signals (written $x = y\, f\, z$). A signal x represents an infinite flow of values. It is sampled according to the discrete pace of its clock, noted $\hat{x}$. The lexical scope of a signal x is restricted to a process P by P/x. An equation $x = y\, f\, z$ defines

the output signal x by the relation of its input signals y and z through the operator f. A process defines the simultaneous solution of the equations it is composed of.

$$P, Q \ ::= \ x = y \, f \, z \mid P \,|\, Q \mid P/x \qquad \text{(process)}$$

As a result, an equation partially relates signals in an abstract timing model, represented by clock relations, and a process defines the simultaneous solution of the equations in that timing model. Signal defines the following kinds of primitive equations:

- A sampling $x = y \,\mathsf{when}\, z$ defines x by y when z is true and both y and z are present. In a sampling equation, the output signal x is present iff both input signals y and z are present and z holds the value *true*.
- A merge $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise. In a merge equation, the output signal is present iff either of the input signals y or z is present.
- A delay equation $x = y \,\mathsf{pre}\, v$ initially defines the signal x by the value v and then by the value of the signal y from the previous execution of the equation. In a delay equation, the signals x and y are assumed to be synchronous, i.e. either simultaneously present or simultaneously absent at all times.
- A functional equation $x = y \, f \, z$ defines x by the successive values of its synchronized operands y and z through the arithmetic or boolean operation f.

In the remainder, we write $\mathcal{V}(P)$ for the set of free signal names x of P (they occur in an equation of P and their scope is not restricted). A free signal is an output iff it occurs on the left hand-side of an equation. Otherwise, it is an input signal.

Example. We define the process filter depicted in Section 1. It receives a boolean input signal y and produces an output signal x every time the value of the input changes. The local signal s stores the previous value of the input y at all times. When y first arrives, s is initialized to true. If y and z differ, z is true and then the value of the output x is true, otherwise it is absent.

$$x = \mathsf{filter}(y) \stackrel{\text{def}}{=} (x = \mathsf{true\ when}\, z \,|\, z = (y \neq s) \,|\, s = y \,\mathsf{pre\ true}\,)\,/sz$$

2.1. Model of computation

The formal semantics of Signal in defined in the polychronous model of computation [13]. The polychronous MoC is a refinement of Lee's tagged signal model [17]. In this model, symbolic tags t or u denote periods in time during which execution takes place. Time is defined by a partial order relation $\leq$ on tags ($t \leq u$ means that t occurs before u). A chain is a totally ordered set of tags and defines the clock of a signal: it samples its values over a series of totally related tags. Events, signals, behaviors and processes are defined as follows:

- an *event* is the pair of a tag $t \in \mathbb{T}$ and a value $v \in \mathbb{V}$
- a *signal* is a function from a *chain* of tags to values
- a *behavior* b is a function from names to signals
- a *process* p is a set of behaviors of same domain
- a *reaction* r is a behavior with one time tag t

Example. The meaning of process filter is denoted by a set of behaviors on the signals x and y. In line one, below, we choose a behavior for the input signal y of the equation. In line two defines the meaning of the local signal s by the previous value of y. Notice that it is synchronous to y (it has the same set of tags). In line three, the local signal z is true at the time tags t_i at which y and s hold different values. It is false otherwise. In line four, the output signal x is defined at the time tags t_i at which the signal z is true, as expected in the previous example.

$$\begin{array}{lccccccc} y \mapsto & (t_1,1) & (t_2,0) & (t_3,0) & (t_4,1) & (t_5,1) & (t_6,0) \\ s \mapsto & (t_1,1) & (t_2,1) & (t_3,0) & (t_4,0) & (t_5,1) & (t_6,1) \\ z \mapsto & (t_1,0) & (t_2,1) & (t_3,0) & (t_4,1) & (t_5,0) & (t_6,1) \\ x \mapsto & & (t_2,1) & & (t_4,1) & & (t_6,1) \end{array}$$

Notations. We introduce the notations that are necessary to the formal exposition of the polychronous model of computation. We write $\mathcal{T}(s)$ for the chain of tags of a signal s and $\min s$ and $\max s$ for its minimal and maximal tag. We write $\mathcal{V}(b)$ for the domain of a behavior b (a set of signal names). The restriction of a behavior b to X is noted $b|_X$ (i.e. $\mathcal{V}(b|_X) = X$). Its complementary $b_{/X}$ satisfies $b = b|_X \uplus b_{/X}$ (i.e. $\mathcal{V}(b_{/X}) = \mathcal{V}(b) \setminus X$). We overload the use of $\mathcal{T}$ and $\mathcal{V}$ to talk about the tags of a behavior b and the set of signal names of a process p.

Synchronous structure. The synchronous structure in polychrony is defined by a partial order that relates behaviors holding the same synchronization relations. Informally, two behaviors b and c are said *clock-equivalent*, written $b \sim c$, iff they are equal up to an isomorphism on tags. For instance,

$$\begin{pmatrix} y\mapsto(t_1,1)(t_2,0)(t_3,0) \\ x\mapsto \quad (t_2,1) \end{pmatrix} \sim \begin{pmatrix} y\mapsto(u_1,1)(u_3,0)(u_5,0) \\ x\mapsto \quad (u_3,1) \end{pmatrix}$$

The synchronization of a behavior b with a behavior c is noted $b \leq c$. It can be defined as the effect of "stretching" its timing structure. A behavior c is *stretches* a behavior b, written $b \leq c$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and there exists a bijection f on tags s.t.

$\forall t, u, t \leq f(t) \wedge (t < u \Leftrightarrow f(t) < f(u))$

$\forall x \in \mathcal{V}(b), \mathcal{T}(c(x)) = f(\mathcal{T}(b(x))) \wedge \forall t \in \mathcal{T}(b(x)), b(x)(t) = c(x)(f(t))$

Two behaviors b and c are said *clock-equivalent*, written $b \sim c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$. The synchronous composition $p \,|\, q$ of two processes p and q is defined by combining behaviors $b \in p$ and $c \in q$ that are identical on $I = \mathcal{V}(p) \cap \mathcal{V}(q)$, the interface between p and q.

$$p \,|\, q = \{b \cup c \,|\, (b,c) \in p \times q \ \wedge \ b|_I = c|_I \ \wedge \ I = \mathcal{V}(p) \cap \mathcal{V}(q)\}$$

Asynchronous structure. Whereas "stretching" informally depicts the timing relations maintained in a synchronous structure, the effect of "relaxing" best described the timing structure of desynchrnized behaviors. For instance, consider two behaviors b and c which carry events at unrelated time tages $t_{1..3}$ and $u_{1..3}$. Since their signals x and y carry the same values in the same order, they are said *flow-equivalent*.

$$b = \begin{pmatrix} y\mapsto(t_1,1)(t_2,0)(t_3,0) \\ x\mapsto \quad (t_2,1) \end{pmatrix} \approx \begin{pmatrix} y\mapsto(u_1,1)(u_2,0)(u_3,0) \\ x\mapsto(u_1,1) \end{pmatrix} = c$$

Formally, a behavior c *relaxes* b, written $b \sqsubseteq c$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and, for all $x \in \mathcal{V}(b)$, $b|_x \leq c|_x$. Two behaviors b and c are *flow-equivalent*, written $b \approx c$, iff there exists a behavior d s.t. $b \sqsupseteq d \sqsubseteq c$.

Alternatively, a relaxed behavior $c \sqsupseteq b$ may be seen as the result of passing all events of the behavior b through a first-in-first-out buffer of either infinite length or of possibly infinite transport time, exactly as if it was communicated through asynchronous composition between two processes p and q.

Hence, the asynchronous composition $p \parallel q$ of two processes p and q is defined by the set of behaviors d that are flow-equivalent to behaviors $b \in p$ and $c \in q$ along the interface $I = \mathcal{V}(p) \cap \mathcal{V}(q)$.

$$p \parallel q = \left\{ d \,|\, (b,c) \in p \times q \;\wedge\; b_{/I} \cup c_{/I} \leq d_{/I} \;\wedge\; b|_I \sqsubseteq d|_I \sqsupseteq c|_I \;\wedge\; I = \mathcal{V}(p) \cap \mathcal{V}(q) \right\}$$

Scheduling structure. To render the causality of events occurring at the same time tag t, we refine the domain of polychrony with a scheduling relation defined on an abstract domain of dates $\mathcal{D}$. A date d consists of a time t and a location x. The relation $t_x \rightarrow u_y$ means that the event along the signal named y at u may not happen before x at t. When no ambiguity is possible on the identity of b in a scheduling constraint, we write it $t_x \rightarrow t_y$. We constrain scheduling $\rightarrow$ to contain causality so that $t < t'$ implies $t_x \rightarrow^b t'_x$ and $t_x \rightarrow^b t'_x$ implies $\neg(t' < t)$.

The definitions for the partial order structure of synchrony and asynchrony in the polychronous model of computation extend point-wise to account for scheduling relations. We say that a behavior c is a *stretching* of b, written $b \leq c$, iff $\mathcal{V}(b) = \mathcal{V}(c)$ and there exists a bijection f on $\mathcal{T}$ which satisfies

$\forall t, t' \in \mathcal{T}(b), t \leq f(t) \wedge (t < t' \Leftrightarrow f(t) < f(t'))$
$\forall x, y \in \mathcal{V}(b), \forall t \in \mathcal{T}(b(x)), \forall t' \in \mathcal{T}(b(y)), t_x \rightarrow^b t'_y \Leftrightarrow f(t)_x \rightarrow^c f(t')_y$
$\forall x \in \mathcal{V}(b), \mathcal{T}(c(x)) = f(\mathcal{T}(b(x))) \wedge \forall t \in \mathcal{T}(b(x)), b(x)(t) = c(x)(f(t))$

Concatenation of reactions. The formulation of the denotational semantics, presented next, and of its formal properties, make extensive use of the notion of reaction and concatenation. A reaction r is a behavior with (at most) one time tag t. We write $\mathcal{T}(r)$ for the tag of a non empty reaction r. An empty reaction of the signals X is noted $\emptyset|_X$. The empty signal is noted $\emptyset$. A reaction r is concatenable to a behavior b iff $\mathcal{V}(b) = \mathcal{V}(r)$, and, for all $x \in \mathcal{V}(b)$, $r(x) = \emptyset$ or $\max(\mathcal{T}b(x)) < \min(\mathcal{T}r(x))$. If so, concatenating r to b is defined by

$$\forall x \in \mathcal{V}(b), \forall u \in \mathcal{T}(b) \cup \mathcal{T}(r), (b \cdot r)(x)(u) = \text{if } u \in \mathcal{T}(r(x)) \text{ then } r(x)(u) \text{ else } b(x)(u)$$

Example. Two reactions of signal-wise related time tags can be concatenated, written $r \cdot s$, to form a behavior. For instance, if $t_1 < t_2$ and $t_2 < t_3$ we can construct the following behavior of the signals x, y at the instants $t_{1,2,3}$ using two successive concatenations. Notice that the extension of concatenation to behaviors is associative.

$$\begin{pmatrix} y{\mapsto}(t_1,1) \\ x{\mapsto} \end{pmatrix} \cdot \begin{pmatrix} y{\mapsto}(t_2,0) \\ x{\mapsto}(t_2,1) \end{pmatrix} \cdot \begin{pmatrix} y{\mapsto} \\ x{\mapsto}(t_3,1) \end{pmatrix} = \begin{pmatrix} y{\mapsto}(t_1,1)(t_2,0) \\ x{\mapsto} \qquad (t_2,1)(t_3,1) \end{pmatrix}$$

2.2. Semantics of Signal

The semantics $[\![P]\!]$ of a Signal process P is defined by a set of behaviors that are inductively constructed by the concatenation of reactions. We assume that the empty behavior on $\mathcal{V}(P)$, noted $\emptyset|_{\mathcal{V}(p)}$, belongs to $[\![P]\!]$, for all P.

The semantics of deterministic merge $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise.

$$[\![x = y \,\mathsf{default}\, z]\!] = \left\{ b \cdot r \,\middle|\, b \in [\![x = y \,\mathsf{default}\, z]\!], r(x) = \begin{vmatrix} r(y), & \text{if } r(y) \neq \emptyset \\ r(z), & \text{if } r(y) = \emptyset \end{vmatrix} \right\}$$

The semantics of sampling $x = y \,\mathsf{when}\, z$ defines x by y when z is true.

$$[\![x = y \,\mathsf{when}\, z]\!] = \left\{ b \cdot r \,\middle|\, \begin{array}{l} b \in [\![x = y \,\mathsf{when}\, z]\!], \\ u = \max(\mathcal{T}(b(y))),\ r(x) = \end{array} \left| \begin{array}{ll} r(y), & \text{if } r(z)(t) = \mathsf{true} \\ \emptyset, & \text{if } r(z)(t) = \mathsf{false} \\ \emptyset, & \text{if } r(z) = \emptyset \end{array} \right. \right\}$$

The semantics of a delay $x = y \,\mathsf{pre}\, v$ initially defines x by the value v (for an initially empty behavior b) and then by the previous value of y (i.e. $b(y)(u)$ where u is the maximal tag of b).

$$[\![x = y \,\mathsf{pre}\, v]\!] = \left\{ b \cdot r \,\middle|\, \begin{array}{l} b \in [\![x = y \,\mathsf{when}\, z]\!], \\ u = \max(\mathcal{T}(b(y))),\ r(x) = \end{array} \left| \begin{array}{ll} t \mapsto b(y)(u), & \text{if } r(y) \neq \emptyset \wedge b \neq \emptyset_{xy} \\ t \mapsto v, \text{if} & r(y) \neq \emptyset \wedge b = \emptyset_{xy} \\ \emptyset, & \text{if } r(y) = \emptyset \wedge b = \emptyset_{xy} \end{array} \right. \right\}$$

The meaning of the synchronous composition $P \,|\, Q$ is defined by $[\![P \,|\, Q]\!] = [\![P]\!] \,|\, [\![Q]\!] = \{p \,|\, q \,|\, (p, q) \in [\![P]\!] \times [\![Q]\!]\}$. The meaning of restriction is defined by $[\![P/x]\!] = \{c \,|\, b \in [\![P]\!] \wedge c \leq (b_{/x})\}$.

Example. The meaning of the equation $x = \mathsf{true} \,\mathsf{when}\, (y \neq (y \,\mathsf{pre}\, \mathsf{true}))$ consists of a set of behaviors with two signals x and y. On line one, below, we choose a behavior for the input signal y of the equation. On line two, we define the signal for the expression $y \,\mathsf{pre}\, \mathsf{true}$ by application of the function $[\![]\!]$. Notice that y and $y \,\mathsf{pre}\, \mathsf{true}$ are synchronous (they have the same set of tags). On line three, the output signal x is defined at the time tags t_i when y and $y \,\mathsf{pre}\, \mathsf{true}$ hold different values, as expected in the previous example.

$y \mapsto$	(t_1, true)	(t_2, false)	(t_3, false)	(t_4, true)	(t_5, true)	(t_6, false)
$y \,\mathsf{pre}\, \mathsf{true} \mapsto$	(t_1, true)	(t_2, true)	(t_3, false)	(t_4, false)	(t_5, true)	(t_6, true)
$x \mapsto$		(t_2, true)		(t_4, true)		(t_6, true)

2.3. Formal properties

The formal properties considered in the remainder pertain to the insensitivity of timing relations in a process p (its local clock relations) to external communication delays. The property of endochrony, Definition 1, guarantees that the synchronization performed by a process p is independent from latency in the network. Formally, let I be a set of input signals of p, whenever the process p admits two input behaviors $b|_I$ and $c|_I$ that are assumed to be flow equivalent (timing relations have been altered by the network) then p always reconstructs the same timing relations in b and c (up to clock-equivalence).

Definition 1. *A process p is* endochronous *iff there exists $I \subset \mathcal{V}(p)$ s.t., for all $b, c \in p$, $b|_I \approx c|_I$ implies $b \sim c$.*

Example. To prove that the filter is endochronous, consider two of its possible traces b and c with flow-equivalent input signals

$$b(y) = (t_1, 1)(t_2, 0)(t_3, 0)(t_4, 1) \text{ and } c(y) = (u_1, 1)(u_2, 0)(u_3, 0)(u_4, 1)$$

They share no tags, but carry the same flow of values. The filter necessarily constructs the output signals

$$b(x) = (t_2, 1)(t_4, 1) \text{ and } c(x) = (u_2, 1)(u_4, 1)$$

One notices that b and c are equivalent by a bijection $(t_i \mapsto u_i)_{0<i<5}$ on tags: they are clock-equivalent. Hence, the filter is endochronous.

The weaker definition of endochrony, presented next, requires a definition of the union, written $r \sqcup s$, of two reactions r and s. We say that two reaction r and s are independent iff they have disjoint domains. Two independent reactions of same time tag t can be merged, as $r \sqcup s$.

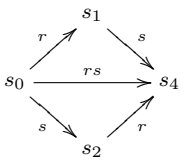
$$\forall x \in \mathcal{V}(r) \cup \mathcal{V}(s), (r \sqcup s)(x) = \text{if } x \in \mathcal{V}(r) \text{ then } r(x) \text{ else } s(x)$$

For instance, $(y \mapsto (t_2, 0)) \sqcup (x \mapsto (t_2, 1)) = (y \mapsto (t_2, 0)x \mapsto (t_2, 1))$.

Definition 2, below, defines the compositional property of weak endochrony in the polychronous model of computation. It is a transposition of Definition 1 in [20]. Informally, a process p is weakly endochronous iff it is deterministic and can perform independent reactions r and s in any order. Note that, by Definition 1, endochrony implies weak-endochrony (e.g. filter is weakly endochronous).

Definition 2. *A process p is* weakly-endochronous *iff*
1. p is deterministic: $\exists I \subset \mathcal{V}(p), \forall b,c \in p,\ b|_I = c|_I \Rightarrow b = c$
2. for all independent reactions r and s, p satisfies:
(a) if $b \cdot r \cdot s \in p$ then $b \cdot s \in p$
(b) if $b \cdot r \in p$ and $b \cdot s \in p$ then $b \cdot (r \sqcup s) \in p$
(c) if $b \cdot (r \sqcup s)$, $b \cdot (r \sqcup t) \in p$ then $b \cdot r \cdot s$, $b \cdot r \cdot t \in p$

Example. Recall the example of the introduction. The diamond shape of the behavior that results of the synchronous composition of the reader and writer processes is that of a weakly endochronous process. Each atomic behavior (e.g. r, the reader) can be scheduled in any order. Furthermore, it will not alter the possibility to perform the other behavior (e.g. s) at any time.



Definition 3. *p and q are* isochronous *iff $p \,|\, q \approx p \parallel q$*

A process p is non-blocking iff it has a path to a stuttering state (characterized by a reaction r) from any reachable state (characterized by a behavior b).

Definition 4. *p is* non-blocking *iff $\forall b \in p, \exists r, b \cdot r \in p$*

In [20], it is proved that weakly endochronous processes p and q are *isochronous* if they are non-blocking (a locally synchronous reaction of p or q yields a globally asynchronous execution $p \parallel q$).

3. Formal analysis

For the purpose of program analysis and program transformation, the control-flow tree and the data-flow graph of multi-clocked Signal specifications are constructed. These data structures manipulate clocks and signal names.

3.1. Clock relations

A clock c denotes a series of instants (a chain of time tags). The clock $\hat{x}$ of a signal x denotes the instants at which the signal x is present. The clock $[x]$ (resp. $[\neg x]$) denotes the instants at which x is present and holds the value true (resp. false).

$$c ::= \hat{x} \,|\, [x] \,|\, [\neg x] \quad \text{(clock)}$$

A clock expression e is either the empty clock, noted 0, a signal clock c, or the conjunction $e_1 \wedge e_2$, the disjunction $e_1 \vee e_2$, the symmetric difference $e_1 \setminus e_2$ of e_1 and e_2.

$$e ::= 0 \,|\, c \,|\, e_1 \wedge e_2 \,|\, e_1 \vee e_2 \,|\, e_1 \setminus e_2 \quad \text{(clock expression)}$$

The meaning $[\![e]\!]_b$ of a clock e is defined with respect to a given behavior b and consists of the set of tags satisfied by the proposition e in the behavior b. The meaning of the clock $x = v$ (resp. $x = y$) in b is the set of tags $t \in \mathcal{T}(b(x))$ (resp. $t \in \mathcal{T}(b(x)) \cap \mathcal{T}(b(y))$) such that $b(x)(t) = v$ (resp. $b(x)(t) = b(y)(t)$). In particular, $[\![\hat{x}]\!]_b = \mathcal{T}(b(x))$ and $[\![[x]]\!]_b = [\![x = \mathsf{true}]\!]_b$. The meaning of a conjunction $e \wedge f$ (resp. disjunction $e \vee f$ and difference $e \setminus f$) is the intersection (resp. union and difference) of the meaning of e and f. Clock 0 has no tags.

$$\begin{array}{lll} [\![1]\!]_b = \mathcal{T}(b) \qquad [\![0]\!]_b = \emptyset & & [\![e \wedge f]\!]_b = [\![e]\!]_b \cap [\![f]\!]_b \\ [\![x = v]\!]_b = \{t \in \mathcal{T}(b(x)) \,|\, b(x)(t) = v\} & & [\![e \vee f]\!]_b = [\![e]\!]_b \cup [\![f]\!]_b \\ [\![x = y]\!]_b = \{t \in \mathcal{T}(b(x)) \cap \mathcal{T}(b(y)) \,|\, b(x)(t) = b(y)(t)\} & & [\![e \setminus f]\!]_b = b[\![e]\!]_b \setminus [\![f]\!]_b \end{array}$$

3.2. Scheduling relations

Signals and clocks are related by synchronization and scheduling relations, denoted R. A scheduling relation $a \rightarrow^c b$ specifies that the calculation of the node b, a signal or a clock, cannot be scheduled before that of the node a when the clock c is present.

$$a, b ::= x \,|\, \hat{x} \quad \text{(node)}$$

A clock relation $c = e$ specifies that the signal clock c is present iff the clock expression e is true. Just as ordinary processes P, relations R are subject to composition $R \,|\, S$ and to restriction R/x.

$$R, S ::= c{=}e \,|\, a \rightarrow^c b \,|\, (R \,|\, S) \,|\, R/x \quad \text{(timing relation)}$$

A scheduling specification $y \rightarrow x$ at clock e denotes the behaviors b on $\mathcal{V}(e) \cup \{x, y\}$ which, for all tags $t \in [\![e]\!]_b$, requires x to preceed y: if t is in $b(x)$ then it is necessarily in $b(y)$ and satisfies $t_y \rightarrow^b t_x$.

$$[\![y \rightarrow^c x]\!] = \{b \,|\, \mathcal{V}(b) = \mathcal{V}(c) \cup \{x, y\} \wedge \forall t \in [\![c]\!]_b,\, t \in \mathcal{T}(b(x)) \Rightarrow t \in \mathcal{T}(b(y)) \wedge t_y \rightarrow^b t_x\}$$

3.3. Clock inference system

The inference system $P : R$ associates a process P with its implicit timing relations R. Deduction starts from the assignment of clock relations to primitive equations and is defined by induction on the structure of P: the deduction for composition $P \,|\, Q$ and for P/x are induced by the deductions $P : R$ and $Q : S$ for P and Q.

$$P : R \wedge Q : S \Rightarrow P \,|\, Q : R \,|\, S \qquad P : R \Rightarrow P/x : R/x$$

In a delay equation $x = y \,\mathsf{pre}\, v$, the input and output signals are synchronous, written $\hat{x} = \hat{y}$, and do not have any scheduling relation.

$$x = y \,\mathsf{pre}\, v : (\hat{x} = \hat{y})$$

In a sampling equation $x = y \,\mathsf{when}\, z$, the clock of the output signal x is defined by that of $\hat{y}$ and sampled by $[z]$. The input y is scheduled before the output when both $\hat{y}$ and $[z]$ are present, written $y \rightarrow^{\hat{x}} x$.

$$x = y \,\mathsf{when}\, z : (\hat{x} = \hat{y} \wedge [z] \,|\, y \rightarrow^{\hat{x}} x)$$

In a merge equation $x = y \,\mathsf{default}\, z$, the output signal x is present if either of the input signals y or z are present. The first input signal y is scheduled before x when it is present, written $y \rightarrow^{\hat{y}} x$. Otherwise z is scheduled before x, written $z \rightarrow^{\hat{z}\setminus\hat{y}} x$.

$$x = y \,\mathsf{default}\, z : (\hat{x} = \hat{y} \vee \hat{z} \,|\, y \rightarrow^{\hat{y}} x \,|\, z \rightarrow^{\hat{z}\setminus\hat{y}} x)$$

A functional equation $x = y \, f \, z$ synchronizes and serializes its input and output signals.

$$x = y \, f \, z : (\hat{x} = \hat{y} = \hat{z} \,|\, y \rightarrow^{\hat{x}} x \,|\, z \rightarrow^{\hat{x}} x)$$

We write $R \models S$ to mean that R satisfies S in the Boolean algebra in which timing relations are expressed: composition $R \,|\, S$ stands for conjunction and restriction R/x for existential quantification (some examples are given below). For all boolean signals x in $\mathcal{V}(R)$, we assume that $R \models \hat{x} = [x] \vee [\neg x]$ and $R \models [x] \wedge [\neg x] = 0$.

Example. To outline the use of clock and scheduling relation analysis in Signal, we consider the specification and analysis of a one-place buffer. Process buffer implements two functionalities: flip and current.

$$x = \mathsf{buffer}(y) \stackrel{\text{def}}{=} (x = \mathsf{current}(y) \,|\, \mathsf{flip}(x, y))$$

The process flip synchronizes the signals x and y to the true and false values of an alternating boolean signal t.

$$\mathsf{flip}(x, y) \stackrel{\text{def}}{=} (s = t \,\mathsf{pre}\, \mathsf{true} \,|\, t = \mathsf{not}\, s \,|\, \hat{x} = [t] \,|\, \hat{y} = [\neg t]) \,/ st$$

The process current stores the value of an input signal y and loads it into the output signal x upon request.

$$x = \mathsf{current}(y) \stackrel{\text{def}}{=} (r = y \,\mathsf{default}\, (r \,\mathsf{pre}\, \mathsf{false}) \,|\, x = r \,\mathsf{when}\, \hat{x} \,|\, \hat{r} = \hat{x} \vee \hat{y}) \,/ r$$

The inference system $P : R$ infers the clock relations that denote the synchronization constraints implied by process buffer. There are four of them:

$$\hat{r} = \hat{s} \qquad \hat{t} = \hat{x} \vee \hat{y} \qquad \hat{x} = [t] \qquad \hat{y} = [\neg t]$$

From these equations, we observe that process buffer has three clock equivalence classes. The clocks $\hat{s}, \hat{t}, \hat{r}$ are synchronous and define the master clock equivalence class of buffer. The two other classes, $\hat{x} = [t]$ and $\hat{y} = [\neg t]$, are samples of the signal t.

$$\hat{r} = \hat{s} = \hat{t} \qquad \hat{x} = [t] \qquad \hat{y} = [\neg t]$$

Together with scheduling analysis, the inference system yields the timing relation R_{buffer} of the process under analysis.

$$R_{\mathsf{buffer}} \stackrel{\text{def}}{=} \left(\hat{x} = [t] \,|\, \hat{y} = [\neg t] \,|\, \hat{r} = \hat{x} \vee \hat{y} \,|\, s \rightarrow^{\hat{s}} t \,|\, y \rightarrow^{\hat{y}} r \,|\, r \rightarrow^{\hat{x}} x\right) /rst$$

From R_{buffer}, we deduce $\hat{r} = \hat{t}$. Since t is a boolean signal, $\hat{t} = [t] \vee [\neg t]$ (a signal is always true or false when present). By definition of R_{buffer}, $\hat{x} = [t]$ and $\hat{y} = [\neg t]$ (x and y are sampled from t). Hence, we have $\hat{r} = \hat{x} \vee \hat{y}$ and can deduce that $R_{\mathsf{buffer}} \models (\hat{r} = \hat{t})$.

3.4. Clock hierarchy

The internal data-structures manipulated by the Signal compiler for program analysis and code generation consists of a clock hierarchy and of a scheduling graph. The clock hierarchy represents the control-flow of a process by a partial order relation. The scheduling graph defines a fine-grained scheduling of otherwise synchronous signals. The structure of a clock hierarchy is denoted by a partial order relation $\preceq$ defined as follows.

Definition 5. *The* hierarchy $\preceq$ *of a process* $P : R$ *is the transitive closure of the maximal relation defined by the following axioms and rules:*

1. *for all boolean signals* x, $\hat{x} \preceq [x]$ *and* $\hat{x} \preceq [\neg x]$
2. *if* $R \models b = c$ *then* $b \preceq c$ *and* $c \preceq b$, *written* $b \sim c$
3. *if* $R \models b_1 = c_1 \, f \, c_2$, $f \in \{\wedge, \vee, \backslash\}$, $b_2 \preceq c_1$, $b_2 \preceq c_2$ *then* $b_2 \preceq b_1$.

We refer to $c_\sim$ *as the clock equivalence class of* c *in the hierarchy* $\preceq$

1. For all boolean signals x of R, define $\hat{x} \preceq [x]$ and $\hat{x} \preceq [\neg x]$. This means that, if we know that x is present, then we can determine whether x is true or false.
2. If $b = c$ is deductible from R then define $b \preceq c$ and $c \preceq b$, written $b \sim c$. This means that if b and c are synchronous, and if either of the clocks b or c is known to be present, then the presence of the other can be determined.
3. If $R \models b_1 = c_1 \, f \, c_2$, $f \in \{\wedge, \vee, \backslash\}$, $b_2 \preceq c_1$, $b_2 \preceq c_2$ then $b_2 \preceq b_1$. This means that if b_1 is defined by $c_1 \, f \, c_2$ in g and if both clocks c_1 and c_2 can be determined once their common upper bound b_2 is known, then b_1 can also be determined when b_2 is known.

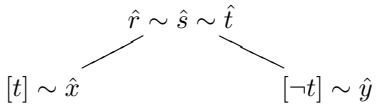
A well-formed hierarchy has no relation $b \preceq c$ that contradicts Definition 5. For instance, the hierarchy of the process $x = y$ and $z \,|\, z = y$ when y is ill-formed, since $\hat{y} \sim [y]$. A process with an ill-formed hierarchy may block.

Definition 6. *A hierarchy* $\preceq$ *is* ill-formed *iff* either $\hat{x} \succeq [x]$ *or* $\hat{x} \succeq [\neg x]$, *for any* x, *or* $b_1 \preceq b_2$ *for any* $b_1 = c_1 \, f \, c_2$ *such that* $c_1 \succeq b_2 \preceq c_2$ *and* $b_2 \preceq b_1$

Example. The hierarchy of the buffer is constructed by application of the first and second rules of Definition 5. Rule 2 defines three clock equivalence classes $\{\hat{r}, \hat{s}, \hat{t}\}$, $\{\hat{x}, [t]\}$ and $\{\hat{y}, [\neg t]\}$.

$$\begin{array}{ccc} & \hat{r} \sim \hat{s} \sim \hat{t} & \\ [t] \sim \hat{x} & & [\neg t] \sim \hat{y} \end{array}$$

Rule 1 places the first class above the two others and yields the following structure



Next, one has to define a proper scheduling of all computations to be performed within each clock equivalence class (e.g. to schedule s before t) and across them (e.g. to schedule x or y before r). This task is devoted to scheduling analysis, presented shortly Section 3.6.

3.5. Disjunctive form

Before to perform scheduling analysis, Signal attempts to eliminate all clocks that are expressed using symmetric difference from the graph g of a process. This transformation consists in rewriting clock expressions of the form $e_1 \setminus e_2$ present in the synchronization and scheduling relations of g in a way that does no longer denote the absence of an event e_2, but that is instead computable from the presence or the value of signals.

Example. In the case of process current, for instance, consider the alternative input r pre false in the first equation:

$$r = y \,\mathsf{default}\, (r \,\mathsf{pre}\, \mathsf{false}\,)$$

Its clock is $\hat{r} \setminus \hat{y}$, meaning that the previous value of r is assigned to r only if y is absent. To determine that y is absent, one needs to relate this absence to the presence or the value of another signal.

In the present case, there is an explicit clock relation in the alternate process: $\hat{y} = [\neg t]$. It says that y is absent iff t is present and true. Therefore, one can test the value of t instead of the presence or absence of y in order to deterministically assign either y or r pre false to r

$$y \rightarrow^{[\neg t]} r \,^{[t]} \leftarrow r \,\mathsf{pre}\, \mathsf{false}$$

In [3], it is shown that the symmetric difference $c \setminus d$ between two clocks c and d has a disjunctive form only if c and d have a common minimum b in the hierarchy $\preceq$ of the process, i.e.,

$$c \succeq b \preceq d$$

We say that the timing relation R is in disjunctive form iff it has no clock expression defined by symmetric difference. The implicit reference to absence incurred by symmetric difference can be defined as $c \setminus d =^{\text{def}} c \wedge \overline{d}$ and can be isolated using the following logical decomposition rules.

- conjunction $\overline{c \wedge d} \stackrel{\text{def}}{=} \overline{c} \vee \overline{d}$ and disjunction $\overline{c \vee d} \stackrel{\text{def}}{=} \overline{c} \wedge \overline{d}$.
- positive $\overline{[x]} \stackrel{\text{def}}{=} \overline{\hat{x}} \vee [\neg x]$ and negative $\overline{[\neg x]} \stackrel{\text{def}}{=} \overline{\hat{x}} \vee [x]$ signal occurrences.

The reference to the absence of a signal x, noted $\overline{\hat{x}}$, is eliminated if (and only if) one of the possible elimination rules applies:

- The "zero" rule: $\hat{x} \wedge \overline{\hat{x}} \stackrel{\text{def}}{=} 0$, because a signal is either present or absent, exclusively.
- The "one" rule: $c \wedge (\hat{x} \vee \overline{\hat{x}}) \stackrel{\text{def}}{=} c$, because the presence or the absence of a signal is subsumed by any clock c.
- The synchrony rule: if $d \sim \hat{x}$ then $\overline{\hat{x}} \stackrel{\text{def}}{=} \overline{d}$, to mean that if $\overline{\hat{x}}$ cannot be eliminated but $\hat{x}$ is synchronous to the clock d, then $\overline{d}$ can possibly be eliminated.

Example. In the case of process current in the example of the buffer one infers that $\hat{x} \succeq \hat{t}$ from $\hat{x} \sim [t]$ and $\hat{t} \preceq \hat{y}$ from $\hat{y} \sim [\neg t]$.

$$\hat{y} \sim [\neg t] \qquad \hat{x} \sim [t] \qquad \hat{r} \sim \hat{t}$$

Hence $\hat{x} \succeq \hat{t} \preceq \hat{y}$. Since, in addition, $\hat{r} \sim \hat{t}$, the symmetric difference $\hat{r} \setminus \hat{y}$ can be interpreted as $[t]$.

Timing relations are in disjunctive form iff they have no clock defined by a symmetric difference relation. For instance, suppose that $d \sim [x]$ and that $c \succeq b \preceq d$. Then, the symmetric difference $c \setminus d$ can be eliminated because it can be expressed with $c \wedge [\neg x]$.

Definition 7. *A process P of timing R and hierarchy $\preceq$ is* well-clocked *iff $\preceq$ is well-formed and R is disjunctive.*

3.6. Scheduling graph

Given the control-flow backbone produced using the hierarchization algorithm and clock equations in disjunctive form, the compilation of a Signal specification reduces to finding a proper way to schedule computations within and across clock equivalence classes. The inference system of the previous section defines the precise scheduling between the input and output signals of process buffer. Notice that t is needed to compute the clocks $\hat{x}$ and $\hat{y}$.

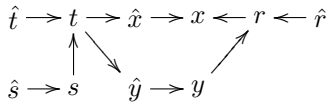
$$s \rightarrow^{\hat{s}} t \qquad y \rightarrow^{\hat{y}} r \qquad r \rightarrow^{\hat{x}} x$$

As seen in the previous section, however, the calculation of clocks in disjunctive form induces additional scheduling constraints, and, therefore, one has to take them into account at this stage. This is done by refining the R with a reinforced one, S, satisfying $S \models R$, and by ordered application of the following rules:

1. $S \models \hat{x} \rightarrow^{\hat{x}} x$ for all $x \in \mathcal{V}(P)$. This means that the calculation of x cannot take place before its clock $\hat{x}$ is known.
2. if $R \models \hat{x} = [y]$ or $R \models \hat{x} = [\neg y]$ then $S \models y \rightarrow^{\hat{y}} \hat{x}$. This means that, if the clock of x is defined by a sample of y, then it cannot be computed before the value of y is known.

3. if $R \models \hat{x} = \hat{y}\, f\, \hat{z}$ with $f \in \{\vee, \wedge\}$ then $S \models \hat{y} \rightarrow^{\hat{y}} \hat{x} \,|\, \hat{z} \rightarrow^{\hat{z}} \hat{x}$. This means that, if the clock of x is defined by an operation on two clocks y and z, then it cannot be computed before these two clocks are known.

Reinforcing the scheduling graph of the buffer yields a refinement of its inferred graph with a structure implied by the calculation of clocks (we just ommitted clocks on arrows to simplify the depiction). Notice that t is now scheduled before the clocks $\hat{x}$ and $\hat{y}$.



Code can be generated starting from this refined structure only if the graph is acyclic. To check whether it is or not, we compute its transitive closure:

1. if $R \models a \rightarrow^{c} b$ then $R \models a \twoheadrightarrow^{c} b$. This just says that the construction of the transitive closure relation $\twoheadrightarrow$ starts from the scheduling graph $\rightarrow$ of the process.
2. if $R \models a \twoheadrightarrow^{c} b$ and $R \models a \twoheadrightarrow^{d} b$ then $R \models a \twoheadrightarrow^{c \vee d} b$. This means that, if b is scheduled after a at both clocks c and d then it is scheduled after a at clock $c \vee d$
3. if $R \models a \twoheadrightarrow^{c} b$ and $R \models b \twoheadrightarrow^{d} z$ then $R \models a \twoheadrightarrow^{c \wedge d} z$. This says that, if b is scheduled after a at clock c and z after b at clock d then z is necessarily scheduled after a at clock $c \wedge d$

The complete graph R of a process P is *acyclic* iff $R \models a \twoheadrightarrow^{e} a$ implies $R \models e = 0$ for all nodes a of R. The graph of our example is.

Definition 8. *A process P of timing relations R is* acyclic *iff the transitive closure $\twoheadrightarrow$ of its scheduling relations R satisfy, for all nodes a, if $a \twoheadrightarrow^{e} a$ then $R \models e = 0$.*

3.7. Sequential code generation

Together with the control-flow graph implied by the timing relations of a process, the scheduling graph is used by Signal to generate sequential or distributed code. To sequentially schedule this graph, Polychrony further refines it in order to remove internal concurrency without affecting its composability with the environment. This is done by observing the following rule.

Definition 9. *The scheduling graph of S reinforces R iff, for any graph T such that $R \,|\, T$ is acyclic, then $R \,|\, S \,|\, T$ is acyclic.*

Starting from a sequential schedule and a hierarchy of process buffer, Polychrony generates simulation code split in several files. The main C file consists of opening the input-output streams of the program, of initializing the value of delayed signals and iteratively executing a transition function until no values are present along the input streams (return code 0). Simulation is finalized by closing the IO streams.

```
int main() {
    bool code;
    buffer_OpenIO();
    code = buffer_initialize();
    while (code) code = buffer_iterate();
    buffer_CloseIO();
}
```

The most interesting part is the transition function. It translates the structure of the hierarchy and of the serialized scheduling graph in C code. It also makes a few optimizations along the way. For instance, r has disappeared from the generated code. Since the value stored in y from one iteration to another is the same as that of r, it is used in place of it for that purpose.

In the C code, the three clock equivalence classes of the hierarchy correspond to three blocks: line 2 (class $\hat{s} \sim \hat{t}$), lines $3-5$ (class $[t] \sim \hat{y}$) and lines $6-9$ (class $[\neg t] \sim \hat{x}$). The sequence of instructions between these blocks follows the sequence $t \rightarrow y \rightarrow x$ of the scheduling graph. Line 10 is the finalization of the transition function. It stores the value that s will hold next time.

```
01. bool buffer_iterate () {
02.     t = !s;
03.     if t {
04.         if !r_buffer_y (&y) return FALSE;
05.     }
06.     if !t {
07.         x = y;
08.         w_buffer_x (x);
09.     }
10.     s = t;
11.     return TRUE;
12. }
```

Also notice that the return code is true, line 11, when the transition function finalizes, but false if it fails to get the signal y from its input stream, line 4. This is fine for simulation code, as we expect the simulation to end when the input stream sample reaches the end. Embedded code does, of course, operate differently. It either waits for y or suspends execution of the transition function until it arrives.

3.8. Endochrony revisited

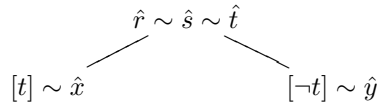
The above code generation scheme yields a way to analyze, transform and execute endochronous specifications. The buffer process, for instance satisfies this property. Literally, it means that the buffer is locally timed. In the transition function of the buffer, this is easy to notice by observing that, at all times, the function synchronizes on either receiving y from its environment or sending x to its environment.

Hence, the activity of the transition function is locally paced by the instants at which the signals x and y are present. However, remember that the structure of control in the transition function is constructed using the hierarchy of process buffer. In the case of an internally timed process, this structure has the particular shape of a tree.

```
if t {
    if !r_buffer_y (&y) return FALSE;
} else {
    x = y; w_buffer_x (x);
}
```

At any time, one can always start reading the state s of the buffer, and calculate t. Then, if t is true, one emits x and, otherwise, one receives y. The presence of any signal

in process buffer is determined from the value of a signal higher in the hierarchy or, at last, from its root.



Formally, regardless of the exact time samples t_1 and t_2 at which it receives an input signal y, or the time samples u_1 and u_2 at which it sends an output signal x, the buffer always behaves according to the same timing relations: t_i occurs strictly before u_i and s is always used at t_i and u_i. The timing relations between the signals x and y of the buffer are independent from latency incurred by communicating with the environment: the buffer is endochronous.

y		t_1		t_2			t'_1				t'_2	
s		t_1	u_1	t_2	u_2	$\sim$	t'_1		u'_1	t'_2		u'_2
x			u_1		u_2				u'_1			u'_2

4. Compositional design criterion

We shall revisit the above schema in light of the compositional design methodology to be presented. To this end, we formulate a decision procedure that uses the clock hierarchy and the scheduling graph of a Signal process to compositionally check the property of isochrony. We start by considering the class of Signal processes P that are reactive and deterministic.

Definition 10. *A process P is* compilable *iff it is well-clocked and acyclic.*

Property 1. *A compilable process P is reactive and deterministic.*

Proof. An immediate consequence of Property 5, in [23], where a well-clocked and acyclic process is proved to be deterministic.

Next, we consider the structure of a compilable Signal specification. It is possibly paced by several, independent, input signals. It necessarily corresponds to a hierarchy $\preceq$ that has several roots. To represent them, we refer to $\preceq^\circ$ as the minimal clock equivalence classes of $\preceq$, and to $\preceq^c$ as the tree of root c in the hierarchy $\preceq$.

$$\preceq^\circ = \{c_\sim \mid c \in \min \preceq\} \qquad \preceq^c = \{(c,d)\} \cup \preceq^d \mid c \preceq d$$

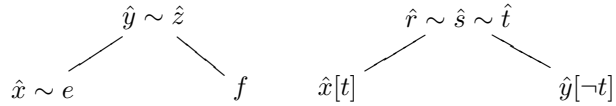
When the hierarchy of a process has a unique root, it is endochronous: the presence of any clock is determined by the presence and values of clocks above it in the hierarchy.

Definition 11. *A process P is* hierarchical *iff its hierarchy has a unique root.*

Property 2. *A compilable and hierarchical process P is endochronous.*

Proof. A detailed proof appears in [23].

Example. The hierarchies of process filter (Section 1), left, and of the buffer, right, are both hierarchical: they are endochronous. Let $e = ([y] \wedge [\neg z]) \vee ([\neg y] \wedge [z])$ and $f = ([\neg y] \wedge [\neg z]) \vee ([y] \wedge [z])$, we have the following hierarchy:



By contrast, a process with several roots necessarily defines concurrent threads of execution. Indeed, and by definition of a hierarchy, its roots cannot be expressed or calculated (or, a fortiori, synchronized or sampled) one with the others. Hence, they naturally define the source of concurrency for the verification of weak endochrony.

4.1. Model checking weak endochrony

Checking that a compilable process p is weakly endochronous reduces to proving that the roots of a process hierarchy satisfy property (2a) of Definition 2 (weak ordering) by using bounded model checking.

Property (2a) can be formulated as an invariant in Signal and submitted to its model checker Sigali [18]. The invariant StateIndependent (x, y) is defined for all pairs of root clock equivalence classes (an abbreviation of the form $[c] = \hat{x}$ stands for $c =$ true when event x default false). It says that, if x is present and y absent at time t (i.e. $cx_t \wedge \neg cy_t$) and if y is present and x absent at time $t+1$ (i.e. $\neg cx_{t+1} \wedge cy_{t+1}$) then x and y can both be present at time t (i.e. $cx_t \wedge cy_t$), written $(\neg cx_t \vee cy_t) \vee (cx_{t+1} \vee \neg cy_{t+1}) \vee (cx_t \wedge cy_t)$.

$$(1)\ i = \mathsf{StateIndependent}\,(x, y) \stackrel{\text{def}}{=} \left(\begin{array}{l} [cx_{t+1}] = \hat{x} \,|\, cx_t = cx_{t+1} \text{ pre false} \\ | \ [cy_{t+1}] = \hat{y} \,|\, cy_t = cy_{t+1} \text{ pre false} \\ | \ \quad i = (\text{ not } cx_t \text{ or } cy_t) \text{ or } (cx_{t+1} \text{ or not } cy_{t+1}) \text{ or } (cx_t \text{ and } cy_t) \end{array} \right) / \begin{array}{l} cx_t, cx_{t+1} \\ cy_t, cy_{t+1} \end{array}$$

Properties (2b-2c) can similarly be checked with the properties OrderIndependent and FlowIndependent. Property OrderIndependent is defined by $(cx_t \wedge \neg cy_t) \wedge (cy_t \wedge \neg cx_t) \Rightarrow (cx_t \wedge cy_t)$. It means that x and y are independently available at all times.

$$(2)\ i = \mathsf{OrderIndependent}\,(x, y) \stackrel{\text{def}}{=} \left(\ [cx_t] = \hat{x} \,|\, [cy_t] = \hat{y} \,|\, i = (\text{ not } cx_t \text{ or } cy_t) \text{ or } (cx_t \text{ or not } cy_t) \text{ or } (cx_t \text{ and } cy_t) \ \right) / cx_t, cy_t$$

Property FlowIndependent is defined for any signal $z \in \mathcal{V}(p)$ by $cz_t \wedge ((cx_t \wedge \neg cy_t) \wedge (cy_t \wedge \neg cx_t)) \Rightarrow cz_t \wedge ((cx_{t+1} \wedge \neg cy_{t+1}) \vee (cy_{t+1} \wedge \neg cx_{t+1}))$.

$$(3)\ i = \mathsf{FlowIndependent}\,(x, y, z) \stackrel{\text{def}}{=} \left(\begin{array}{l} [cx_{t+1}] = \hat{x} \ | \ cx_t = cx_{t+1} \text{ pre false} \\ | \ [cy_{t+1}] = \hat{y} \ | \ cy_t = cy_{t+1} \text{ pre false} \\ | \ [cz_{t+1}] = \hat{z} \ | \ cz_t = cz_{t+1} \text{ pre false} \\ | \ i = (\text{ not } cz_t \text{ or } ((\text{ not } cx_t \text{ or } cy_t) \text{ or } (cx_{t+1} \text{ or not } cy_{t+1}))) \\ \qquad \text{or } (cz_t \text{ and } ((cx_{t+1} \text{ and not } cy_{t+1}) \text{ or } (\text{ not } cx_{t+1} \text{ and } cy_{t+1}))) \end{array} \right) / \begin{array}{l} cx_t, cx_{t+1} \\ cy_t, cy_{t+1} \\ cz_t, cz_{t+1} \end{array}$$

When the clock hierarchy of a compilable process P consists of multiple roots, we can use the above properties to verify that it is weakly endochronous.

Property 3. *A compilable process P whose roots satisfy criteria (1-3) is weakly endochronous.*

Proof. We observe that the formulation of properties (1−3) directly translate Definition 2 in terms of timed Boolean equations. Since they are expressed in Signal, one can model-check them against the specification of the process P under consideration to verify that it is weakly endochronous.

4.2. Static checking isochrony

While the model-checking proposed in Section 4.1 effectively translates Definition 2 to determine the largest possible class of weakly-endochronous processes, it may be sensed to expensive for integration in a design process whose main purpose is automated code generation. In the aim of efficiently generating sequential or concurrent code starting from weakly endochronous specifications, we would like to define a simple and cost-efficient criterion to allow for a large and easily identifiable class of weakly endochronous programs to be statically checked and compiled. To this end, we define the following formal design methodology.

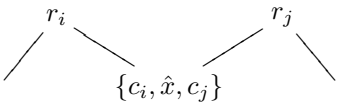
Definition 12. *If P is compilable and hierarchical then it is* weakly hierachic. *If P and Q are weakly hierarchical, $P \,|\, Q$ is well-clocked and acyclic then $P \,|\, Q$ is weakly hierarchical.*

By induction on its structure, a process P is weakly hierarchical iff it is compilable and its hierarchy has roots $r_{1..n}$ such that, for all $1 \leq i < n$, $X_i = \mathcal{V}(\preceq^{r_i})$, $P_i = P|_{X_i}$ is weakly hierarchical and the pair $(\prod_{j=1}^{i} P_j, P_{i+1})$ is well-clocked and acyclic.

Theorem 1.
1. A weakly hierarchical process P is weakly endochronous.
2. If P, Q are weakly hierarchical and $P \,|\, Q$ is well-clocked and acyclic then P and Q are isochronous.

Proof.

1. By definition, a weakly hierarchical process P consists of the composition of a series of processes P_i that are individually compilable and hierarchical, hence endochronous. Since endochrony implies weak endochrony, and since weak endochrony is preserved by composition, the composition P of the P_is is weakly endochronous.
2. Consider the hierarchy of any pair of endochronous processes P_i and P_j in $P \,|\, Q$ that share a common signal x of clock $\hat{x}$. The processes P_i and P_j have roots r_i and r_j and synchronize on $\hat{x}$ at a sub-clock c_i, computed using r_i (since P_i is hierarchical) and at a clock c_j, computed using r_j (since P_j is hierarchical).



   Since $P_i \,|\, P_j$ is well-clocked, the clocks c_i, c_j and hence $\hat{x}$ have a disjunctive form. Hence, it cannot be the case that $\hat{x}$ is defined by the symmetric difference of a clock under r_i and another (e.g. under r_j). Therefore, any reaction initiated in P_i to produce $\hat{x}$ can locally and deterministically decide to wait for a rendez-vous with a reaction of P_j consuming $\hat{x}$. Since P_i and P_j are well-formed, then it cannot be

the case that $\hat{x} = 0$, which would mean that the rendez-vous would never happen. Finally, since $P_i \,|\, P_j$ is acyclic, the rendez-vous of c_i and c_j cannot deadlock. This holds for any pair of endochronous processes P_i and P_j in $P \,|\, Q$, hence $P \,|\, Q$ is non–blocking.

3. These conditions precisely correspond to the weak isochrony criterion of [20], namely, that non-blocking composition (2) of weakly endochronous processes (1) is isochronous. Consequently, the composition of P and Q is isochronous.

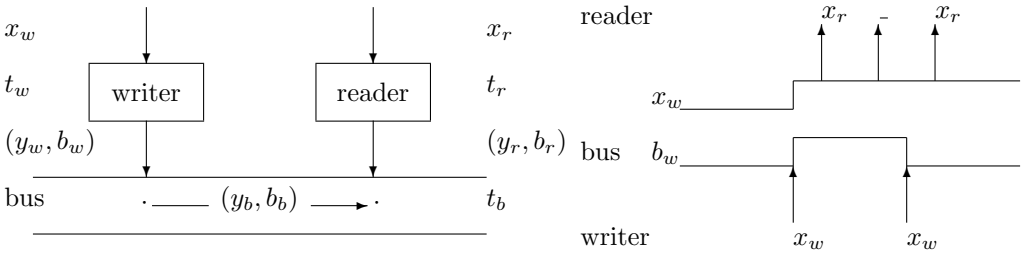
5. A compositional design methodology

Our method based on model-checking considers the finite-state abstraction of a process where control is expressed by a process on boolean signals and where computation is expressed by a process on infinite value domains (e.g. integers). Hence, it is not exact, and determining that a process is weakly endochronous is indeed not decidable in general for infinite-state systems. The main drawback of this method is that its computational complexity makes it unaffordable for purposes such as program transformation or code generation.

By contrast, our method based on the static abstraction and analysis of clock and scheduling relations reuses the services that our tool implements to perform the successive specification refinement of program transformations from an initial specification to generated code. Its use is of very little complexity overhead but it is less precise, and will potentially reject some programs that may be proved weakly-endochronous using model checking.

Our static criterion for checking the composition of endochronous processes isochronous defines an effective and cost-efficient method for the integration of synchronous modules in the aim of architecture exploration or simulation. Interestingly, this formal methodology meets most of the engineering practice and industrial usage of Signal: the real-time simulation of embedded architectures (e.g. integrated modular avionics) starting from heterogeneous functional blocks (endochronous data-flow functions) and architecture service models (e.g. [14]).

Example of a loosely time-triggered architecture. We consider a simple yet realistic case study built upon the examples we previously presented. We wish to design a simulation model for a loosely time-triggered architecture (LTTA) [4]. The LTTA is composed of three devices, a *writer,* a *bus,* and a *reader.* Each device is paced by its own clock.



At the nth clock tick (time $t_w(n)$), the *writer* generates the value $x_w(n)$ and an alternating flag $b_w(n)$. At any time $t_w(n)$, the writer's output buffer (y_w, b_w) contains

the last value that was written into it. At $t_b(n)$, the *bus* fetches (y_w, b_w) to store in the input buffer of the reader, denoted by (y_b, b_b). At $t_r(n)$, the *reader* loads the input buffer (y_b, b_b) into the variables $y_r(n)$ and $b_r(n)$. Then, in a similar manner as for an alternating bit protocol, the reader extracts $y_r(n)$ iff $b_r(n)$ has changed.

A simulation model of the LTTA. To model an LTT architecture in Signal, we consider two data-processing functions that communicate by writing and reading values on an LTT bus. In Signal, we model an interface of these functions that exposes their (limited) control. The writer accepts an input x_w and defines the boolean flag b_w that is carried along with it over the bus.

$$(y_w, b_w) = \mathsf{writer}(x_w, c_w) \stackrel{\text{def}}{=} \left(\hat{x}_w = \hat{b}_w = [c_w] \,|\, y_w = x_w \,|\, b_w = \mathsf{not}\,(b_w \;\mathsf{pre}\; \mathsf{true})\right)$$

The reader loads its inputs y_r and b_r from the bus and filters x_r upon a switch of b_r.

$$x_r = \mathsf{reader}(y_r, b_r, c_r) \stackrel{\text{def}}{=} (x_r = y_r \;\mathsf{when}\; \mathsf{filter}(b_r) \,|\, \hat{y}_r = [c_r])$$

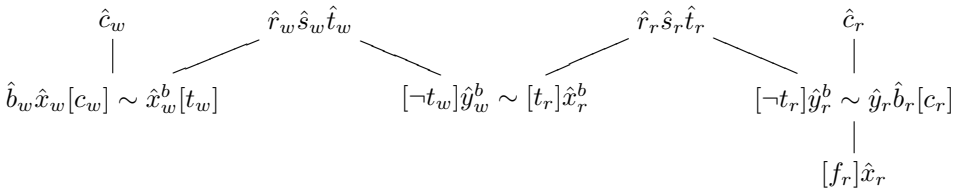
The bus buffers and forwards the inputs y_w and b_w to the reader. The clock c_b is not used since the buffers have local clocks.

$$(y_r, b_r) = \mathsf{bus}(y_w, b_w, c_b) \stackrel{\text{def}}{=} ((y_r, b_r) = \mathsf{buffer}\,(\mathsf{buffer}(y_w, b_w)))$$

The process ltta is defined by its three components reader, bus and writer.

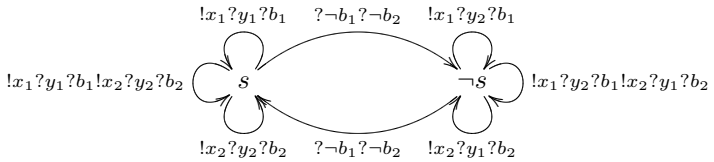
$$x_r = \mathsf{ltta}(x_w, c_w, c_b, c_r) \stackrel{\text{def}}{=} (x_r = \mathsf{reader}\,(\mathsf{bus}\,(\mathsf{writer}\,(x_w, c_w)\,, c_b)\,, c_r))$$

We observe that the hierarchy of the LTTA is composed of four trees. Each tree corresponds to an endochronous and separately compiled process, connected to the other at four rendez-vous points (depicted by equivalence relations $\sim$). The LTTA itself is not endochronous, but it is isochronous because its four components are endochronous and their composition is well-clocked and acyclic.



Static checking vs. model checking. As demonstrated by the example of the LTTA, our static-checking criterion is very-well suited to check isochrony of large systems made by composing endochronous modules. However, some of these modules may not strictly be endochronous, as required by Property 2, but still be weakly-endochronous, in the sense of Definition 2, and, unfortunately, admit no decomposition into endochronous sub-modules. An example of such a process is the crossbar switch presented in [20]. Its specification consists of an automaton that switches the routes of two input signals (y_1, y_2) along two output signals (x_1, x_2) depending on the values of two control signals

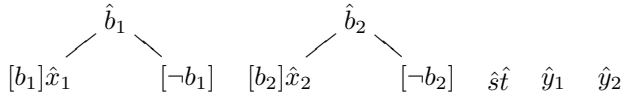
(b_1, b_2). Signal x_1 (resp. x_2) is output iff b_1 is present and true (resp. b_2). Its value is y_1 in state s and y_2 in state $\neg s$ (resp. y_2 or y_1).



The specification of the switch in Signal consists of data-flow equations (for x_1, x_2), state transitions (equations for the state s and its previous value t) and synchronization constraints (for $\hat{x}_1, \hat{x}_2, \hat{y}_1, \hat{y}_2$). The automaton synchronizes the presence of x_i with b_i true, hence $\hat{x}_i = [b_i]$. It performs state transitions when b_1, b_2 are false, hence $s = \neg t$ when $\neg b_1$ and when $\neg b_2$ (and $s = t$ otherwise). Finally, each input signal y_i is present iff either b_i and s are true or $b_{j\neq i}$ is true and s is false.

$$(x_1, x_2) = \mathsf{switch}\,(y_1, b_1, y_2, b_2) \stackrel{\text{def}}{=}$$
$$\left(\begin{array}{l} x_1 = y_1 \,\mathsf{when}\, s \,\mathsf{default}\, y_2 \,\mathsf{when}\, \mathsf{not}\, s \,|\, \hat{x}_1 = [b_1] \,|\, \hat{y}_1 = ([b_1] \wedge [s]) \vee ([b_2] \wedge [\neg s]) \\ | \; x_2 = y_2 \,\mathsf{when}\, s \,\mathsf{default}\, y_1 \,\mathsf{when}\, \mathsf{not}\, s \,|\, \hat{x}_2 = [b_2] \,|\, \hat{y}_2 = ([b_2] \wedge [s]) \vee ([b_1] \wedge [\neg s]) \\ | \; s = (\,\mathsf{not}\, t \,\mathsf{when}\, (\,\mathsf{not}\, b_1 \,\mathsf{when}\, \mathsf{not}\, b_2)) \,\mathsf{default}\, (t \,\mathsf{when}\, (b_1 \,\mathsf{default}\, b_2)) \\ | \; t = s \,\mathsf{pre}\, \mathsf{true} \end{array}\right) /st$$

If we build the hierarchy of the switch, we first observe that it is not endochronous (because its hierarchy is not a tree). Indeed, none of the equations with which clocks $\hat{s}, \hat{t}, \hat{y}_1, \hat{y}_2$ are defined have a common root: they are all defined from $\hat{b}_1$ and $\hat{b}_2$, which are not related. We further observe that none of the trees it is composed corresponds to a subset of equation in the specification. Therefore, it cannot be proved weakly hierarchic in the sense of Definition 12 because it does not have a decomposition into endochronous sub-modules. Fortunately, Property 3 can be model-checked to prove that the switch is indeed weakly-endochronous.



A methodological guideline that can be drawn from the examples of the LTTA and of the switch is that,

1. in most cases, synchronous modules are, just like the reader, writer and bus models of the LTTA, simple data-flow functions that are designed and compiled separately, for which endochrony can easily be checked using Property 2;

2. in some cases, synchronous modules specify complex and control-dominated behavior which, if at all deterministic, can separately be checked weakly endochronous using Property 3;

3. and, finally, the composition of synchronous modules from each of the above categories can compositionally be checked isochronous using Theorem 2.

In a more recent article [21], we propose an alternative to Property 3 for checking modules such as the switch weakly endochronous. It consists of a static analysis of the

Signal specification that determines its minimal set of atomic synchronization patterns (i.e. from which all its possible reaction are constructed). By using this method, it is possible to check that the static abstraction of the switch (that which abstracts the delay equation $t = s \,\mathsf{pre}\, \mathsf{true}$ by $\hat{s} = \hat{t}$) is indeed weakly endochronous.

6. Related Work

In synchronous design formalisms, the design of an embedded architecture is achieved by constructing an endochronous model of the architecture and then by automatically synthesizing ad-hoc synchronization protocols between the elements of this model that will be physically distributed. This technique is called desynchronization and a thorough survey on it is presented in [15]. In the case of Signal, automated distribution is proposed by Aubry [2]. It consists of partitioning endochronous specifications and synthesizing inter-partition protocols to ensure preservation of endochrony.

In [16], Girault et al. propose a different approach for the synchronous languages Lustre and Esterel. It consists in replicating the generated code of an endochronous specification and in replacing duplicated instructions by inter-partition communications. As it uses notions of bi-simulation to safely eliminate blocks, it leads to the construction of a distributed program that consists of endochronously connected programs. But again, distributed code generation is also driven by the global preservation of endochrony.

In [20], the so-called property of weak endochrony is proposed. Weak endochrony supports the compositional construction of globally asynchronous system by adhering to a global objective of weak-isochrony. In [22], we propose an analysis of Signal programs to check this property. However, we observe that it is far more costly than necessary, at least for code generation purposes, as it requires an exhaustive state-space exploration. In [11], Dasgupta et al. also propose a technique to synthesize delay-insensitive protocols for synchronous circuits described with Pétri Nets.

In the model of latency-insensitive protocols [7], components are denoted by the notion of *pearl* ("intellectual property under a shell"). A pearl is required to satisfy an invariant of *patience* (which, in turn, implies endochrony [23]) and a *latency-insensitive protocol* wraps the pearl with a generic client-side controller: a so-called relay station.

The relay station ensures the functional correctness of the pearl by guaranteeing the preservation of signal flows (i.e. isochrony). It implements this function by suspending the pearl's incoming traffic as soon as it is reported to exceed its consumption capability. A technique proposed by Casu et al. in [10] refines this protocol to prevent unnecessary traffic suspension by controlling traffic through pre-determined periodic schedules.

The latency-insensitive protocol is a compositional approach, and can be seen as a "black-box" approach, in that no knowledge on the pearl (but its capability to be patient) is required. Just as desynchronization, Casu's variant [10] is a "grey-box" approach, where knowledge of the pearl is needed to synthesize an an-hoc controller and, at the same time, ensure functional correctness.

Our method defines a class of process that can equally be embedded in a synchronous MoC or in an asynchronous MoC. Therefore, it definitely relates to a larger spectrum of MoCs, such as the SDF and FSMs found in Ptolemy [6], such as Kahn Process Networks [1], and programming paradigms, such as Shim [12]. By contrast with these, our method attempts to take benefits from both the synchronous world, by locally ensur-

ing the highest degree of safety for individual modules, and the asynchronous world, by providing a similar degree of flexibility gained from global compositionality properties.

Our results based on the static method we initially proposed in [24], to which the present article adds a formal proof for Theorem 1. The abstraction defined by the static-checking criterion of Definition 12 defines a simple and effective method to allow for large systems (consisting of many endochronous modules) to be checked weakly endochronous.

However, this abstraction or approximation comes at the cost of rejecting modules which cannot be decomposed into endochronous sub-modules. Some of these modules may however be weakly endochronous, like the crossbar of Section 5. Fortunately, the model-checking criterion of Property 3can instead be used to allow for integrating such modules. As a result, a method to cover the largest possible class of weakly endochronous systems would consist of:

1. checking elementary modules endochronous (using Property 2);
2. checking non-endochronous modules weakly endochronous (using Property 3); and
3. checking the composition of such modules isochronous (using Theorem 2).

7. Conclusions

The clock analysis at the core of our approach shares similarities with desynchronization and latency insensitivity. It avoids the need for any explicit suspension mechanism thanks to the determination of precise timing relations. This yields a cost-effective methodology for the compositional design of globally asynchronous architectures starting from synchronous modules.

This methodology balances a trade-off between cost (of verification) and compositionality (of design). It maintains a compositional global design objective of isochrony while preserving properties secured locally (endochrony) by checking that composition is non-blocking. This yields an efficient approach to compositional modeling embedded architectures which, in addition, meets actual industrial usage.

The commercial implementation of Signal, Sildex, commercialized by TNI, is widely used for the real-time simulation of embedded architectures starting from heterogeneous, possibly foreign, functional blocks (merely endochronous, data-flow functions) and architecture service models (e.g. the ARINC 653 real-time operating system [14]). As an example, TNI has developed a real-time, hardware in-the-loop, simulator of onboard electronic equipments for a car manufacturer.

Our technique efficiently reuses most of existing compilation tool-suites available for Signal in order to implement our proposal, which justifies presenting it in sufficient details in the present article. We are currently upgrading the Polychrony toolset, that supports the Signal specification formalism, with a simple controller-synthesis and code generation scheme supporting the present methodology.

References

[1] Samson Abramsky. A generalized Kahn principle for abstract asynchronous networks. In *International Conference on Mathematical Foundations of Programming Semantics*. Lectures Notes in Computer Science v. 442. Springer, 1989.

[2] Pascal Aubry. Mises en oeuvre distribuées de programmes synchrones. Thèse de l'Université de Rennes, 1997.

[3] Loïc Besnard. Compilation de Signal: horloges, dépendances, environnements. Thèse de l'Université de Rennes, 1992.
[4] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Stravos Tripakis, Jean-Pierre Talpin. A protocol for loosely time-triggered architectures. Embedded Software Conference. Lectures Notes in Computer Science. Springer Verlag, October 2002.
[5] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 2003.
[6] Joseph Buck, Soonhoi Ha, Edward Lee, David Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. The Morgan-Kaufmann Syetems on Silicon series. Kluwer, 2001.
[7] Luca Carloni, Ken McMillan, and Alberto Sangiovanni-Vincentelli. The theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 20(9). IEEE, 2001.
[8] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs and John Plaice. Lustre: a declarative language for programming synchronous systems. *Principles of Programming Languages*. ACM, 1987.
[9] Paul Caspi, Alain Girault, and Daniel Pilaud. Distributing Reactive Systems. International Conference on Parallel and Distributed Computing Systems. ISCA, 1994.
[10] Mario Casu, Luca Macchiarulo. A new approach to latency insensitive design. Design Automation Conference. ACM, 2004.
[11] Sohini Dasgupta, Dumitru Potop-Butucaru, Benoît Caillaud, Alex Yakovlev. Moving from Weakly Endochronous Systems to Delay-Insensitive Circuits. In *Formal Methods for GALS Design*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.
[12] Stephen Edwards, Olivier Tardieu. Shim: a deterministic model for heterogeneous systems. *International Conference on Embedded Software*. ACM, 2005.
[13] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*. World Scientific, 2003.
[14] Abdoulaye Gamatié, Thierry Gautier. Synchronous Modeling of Avionics Applications using the SIGNAL Language. Real-Time and Embedded Technology and Applications Symposium. IEEE, 2003.
[15] Alain Girault. A survey of automatic distribution methods for synchronous programs. In International Workshop on Synchronous Languages, Applications and Programs. Electronic Notes in Theoretical Computer Science. Elsevier, 2005.
[16] Alain Girault, Xavier Nicollin, and Marc Pouzet. Automatic rate desynchronization of embedded reactive programs. ACM Transactions on Embedded Computing Systems, 5(3). ACM, 2006.
[17] Edward Lee, and Alberto Sangiovanni-Vincentelli. "A framework for comparing models of computation". In *IEEE transactions on computer-aided design*, v. 17, n. 12. IEEE, 1998.
[18] Hervé Marchand, Eric Rutten, Michel Le Borgne and M. Samaan. Formal Verification of programs specified with Signal : application to a power transformer station controller. Science of Computer Programming, v. 41(1). Elsevier, 2001.
[19] Julien Ouy, Jean-Pierre Talpin, Loïc Besnard, and Paul Le Guernic. Separate compilation of polychronous specifications. *Formal Methods for Globally Asynchronous Locally Synchronous Design*. Electronic Notes in Theoretical Computer Science, Elsevier, 2007.
[20] Dimitru Potop-Butucaru, Benoit Caillaud, and Albert Benveniste. Concurrency in synchronous systems. In *Formal Methods in System Design*. Kluwer, 2006.
[21] Dimitru Potop-Butucaru, Robert de Simone, Yves Sorel, Jean-Pierre Talpin. From Concurrent Multiclock Programs to Deterministic Asynchronous Implementations. In *Application of Concurrency to System Design*. IEEE Press, 2009.
[22] Jean-Pierre Talpin, Dimitru Potop-Butucaru, Julien Ouy, and Benoit Caillaud. From multi-clocked synchronous specifications to latency-insensitive systems. In *Embedded Software Conference*. ACM, 2005.
[23] Jean-Pierre Talpin and Paul Le Guernic. An algebraic theory for behavioral modeling and protocol synthesis in system design. Formal Methods in System Design. Special Issue on formal methods for GALS design. Springer, 2006.
[24] Jean-Pierre Talpin, Julien Ouy, Loïc Besnard, Paul Le Guernic. Compositional design of isochronous systems. In *Design Analysis and Test in Europe (DATE'08)*. IEEE, 2008.