Compositional design of isochronous systems

Jean-Pierre Talpin, Julien Ouy, Loïc Besnard, Paul Le Guernic

![INRIA]

Compositional design of isochronous systems

Jean-Pierre Talpin — Julien Ouy — Loïc Besnard — Paul Le Guernic

N° 6227 — version 2

initial version June 2007 — revised version November 2007

Rapport de recherche

Compositional design of isochronous systems

Jean-Pierre Talpin, Julien Ouy, Loïc Besnard, Paul Le Guernic

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

Key-words: formal methods, embedded systems, program analysis, synchronous paradigm

Mise en œuvre compositionelle de systèmes isochrones

Résumé : Le paradigme synchrone met en œuvre des méthodes formelles permettant de garantir la correction d'un programme en faisant peu d'hypothèse sur son environnement (d'exécution). Cette correction est assurée par la propriété d'endochronie: l'insensibilité du temps logique, dans le programme, au temps r 'eel, dans l'environnement. L'endochronie peut être vérifiée par analyse statique. elle donne donc facilement et rapidement l'assurance qu'un programme est correct. Malheureusement, elle n'est pas stable par composition, cela complique donc son utilisation dans le cas d'une conception modulaire de programmes. La compositionalié peut cependant être obtenue en affaiblissant la propriété d'endochronie, mais au prix d'une mise en œuvre plus coûteuse car nécéssitant l'exploration de l'espace détats du programme. Cela nous confronte à un dilème entre performance et précision, entre simplicité et compositionalité. Notre objectif est d'aller au delà de cet équilibre en proposant une méthodologie de conception fondée sur l'objectif global de composer des modules non-blocants et sur l'objectif local d'assurer l'endochronie (de chaque module). La propriété obtenue est compositionelle et le coût de sa vérification est faible.

Mots-clés : méthodes formelles, systèmes embarqués, analyse de programmes, paradigme synchrone

1 Introduction

The synchronous paradigm to embedded system design provides strong execution correctness guarantees while requiring minimal assumptions on the execution environment. In most synchronous formalisms, this is achieved by locally verifying that computation (in the system) is insensitive to communication delays (from the environment), i.e., that the system is endochronous ("time is defined from inside").

Example Process filter emits x every time the value of its input y changes. Output tags $t_{2,4}$ are timely related to input tags $t_{1..4}$: Process filter is endochronous.

$$x : (t_1, 1)\ (t_2, 0)\ (t_3, 0)\ (t_4, 1) \quad \rightarrow \boxed{y = \mathsf{filter}(x)} \rightarrow y : \qquad (t_2, 1) \qquad (t_4, 1)$$

In the data-flow formalism Signal, for instance, design is driven by the safety objective of endochrony: endochrony guarantees a synchronization of computations and communications that is independent of possible network latency. Unfortunately, endochrony is not a compositional property: it is not preserved by synchronous composition.

Example The synchronous composition of filter with an endochronous merge equation (to mean "d equals if c then y else z") is no longer endochronous: timing of the output d is not related to one of the inputs c and y.

$$
\begin{array}{l}
c : (t_0, 0)\quad (t_2, 1)\quad (t_4, 1)\quad (t_7, 0) \\
y : \qquad\quad\ (t_2, 1)\quad (t_4, 1) \\
z : (t_0, 1)\qquad\qquad\qquad\quad (t_7, 0)
\end{array}
\ \rightarrow \boxed{d = \mathsf{merge}(c, y, z)} \rightarrow d : (t_0, 1)\quad (t_2, 1)\quad (t_4, 1)\quad (t_7, 0)
$$

In [18], it is shown that compositionality can be achieved by weakening the objective of endochrony: a weakly endochronous system is a deterministic system that can perform independent communications in any order as long as this does not alter its state (i.e. it satisfies the diamond property). It is further shown that the non-blocking composition of weakly endochronous processes is isochronous.

Example The untimed asynchronous composition of processes filter and merge is isochronous: synchronous and asynchronous compositions yield the same flow of values.

$$
\begin{array}{l}
x : 1\ 0\ 0\ 1 \\
c : 0\ 1\ 1\ 0 \\
z : 1\ 0\ 1\ 0
\end{array}
\ \rightarrow \boxed{x = \mathsf{filter}(y)\ \|\ d = \mathsf{merge}(c, y, z)} \rightarrow \quad d : 1\ 1\ 1\ 0
$$

However, 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 a tradeoff between performance (incurred by state-space exploration) and

flexibility (gained from compositionality). We aim at balancing 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 endochronous components).

Our approach consists in globally maintaining a compositional design objective (non-blocking composition) while preserving properties secured locally (endochrony). This yields a less general 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 support the present design methodology, we have designed a simple controller synthesis and code generation scheme [16].

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. It is applied to the exposition of a concurrent code generation scheme in Section 5. 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 $\hat{x}$. 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 functional equation $x = y\,f\,z$ defines an arithmetic or boolean relation f between its operands y, z and the result x.

- 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 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.

The process P/x restricts the lexical scope of the signal x to the process P. 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 z holds the previous value of the input y at all times. When y first arrive, z is initialized to true. If y and z differ then the output x is true, otherwise it is absent.

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

2.1 Model of computation

The formal semantics of Signal in defined in the polychronous model of computation [9]. The polychronous MoC is a refinement of Lee's tagged signal model [14]. 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. Line one, below, we choose a behavior for the input signal y of the equation. Line two defines the meaning of the local signal z by the previous value of y. Notice that it is synchronous to y (it has the same set of tags). Line three, the output signal x is defined at the time tags t_i at which y and z hold different values, as expected in the previous example.

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

Notations   We introduce the notations that are necessary to the formal exposition of the poly-chronous 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.

Synchrony and asynchrony   Informaly, two behaviors b and c are *clock-equivalent*, written $b \sim c$, iff they are equal up to an isomorphism on tags. For instance,

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

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

$\quad \forall t, u, t \leq f(t) \wedge (t < u \Leftrightarrow f(t) < f(u))$
$\quad \forall x \in \mathcal{V}(b), \mathcal{T}(c(x)) = f(\mathcal{T}(b(x))) \wedge \forall t \in \mathcal{T}(b(x)), b(x)(t) = c(x)(f(t))$

b and c are *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.

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

Asynchrony   Similarly, two behaviors b and c are *flow-equivalent*, written $b \approx c$, iff they have the same domain and all signals carry the same values in the same order. For instance,

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

Desynchronization is defined as the effect of "relaxing" the timing structure of a behavior: a behavior c is a *relaxation* of 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. $b \sqsupseteq d \sqsubseteq c$. The asynchronous composition $p \,\|\, q$ 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 $I = \mathcal{V}(p) \cap \mathcal{V}(q)$.

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

Concatenation   The semantics $[\![P]\!]$ of a Signal process P, presented next, is defined by a set of behaviors that are inductively constructed by the concatenation of reactions. 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 $\varnothing|_X$. 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 $x \in \mathcal{V}(b)$, $\max(b(x)) < \mathcal{T}(r(x))$. If so, concatenating r to b is defined by

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

Example Two reactions of signal-wise related time tags can be concatenated, written $r \cdot s$, to form a behavior. For instance,

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

2.2 Semantics of Signal

The semantics $[\![P]\!]$ of a Signal process P is a set of behaviors that are inductively defined by the concatenation of reactions.

Initially, we assume that $\varnothing|_{\mathcal{V}(p)} \in [\![P]\!]$. The semantics of a delay $x = y\,\mathsf{pre}\,v$ is defined by appending a reaction r of tag t to a behavior b. It initially defines x by the value v (when b is empty) and then by the previous value of y (i.e. $b(y)(u)$ where u is the maximal tag of b).

$$[\![x = y\,\mathsf{pre}\,v]\!] = \left\{ b \cdot r \,\middle|\, \begin{matrix} b \in [\![x = y\,\mathsf{when}\,z]\!], \\ u = \max(\mathcal{T}(b(y))), \; r(x) = \\ t = \mathcal{T}(r), \end{matrix} \,\middle|\, \begin{matrix} t \mapsto b(y)(u), & r(y) \neq \emptyset \wedge b \neq \varnothing_{xy} \\ t \mapsto v, & r(y) \neq \emptyset \wedge b = \varnothing_{xy} \\ \emptyset, & r(y) = \emptyset \wedge b = \varnothing_{xy} \end{matrix} \right\}$$

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

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

Finally, $x = y\,\mathsf{default}\,z$ defines x by y when y is present and by z otherwise.

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

The meaning of the synchronous composition $P\,|\,Q$ is the synchronous composition $[\![P\,|\,Q]\!] = [\![P]\!]\,|\,[\![Q]\!]$ of the meaning of P and 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.

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

Formal properties   The formal properties considered in the remainder pertain 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 check that the filter is endochronous, consider two of its possible trace b and c with flow-equivalent input signals $b(y) = (t_1, 1)(t_2, 0)(t_3, 0)(t_4, 1)$ 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 $b(x) = (t_2, 1)(t_4, 1)$ 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. This is no longer the case if it is composed with process merge.

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 $r \sqcup s$.

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

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. Informally, 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   For instance, the synchronous composition of processes filter and merge is weakly endochronous: it is deterministic and all combinations of reactions consisting of the signals x, y, z and c belong to its possible behaviors.

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

A process p is non-blocking iff, in any reachable state (characterized by a behavior b), it has a path to a stuttering state (characterized by a reaction r). Notice that the composition of filter and merge is non-blocking.

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

In [18], 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 and scheduling 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).

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

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.

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

Signals and clocks are related by synchronization and scheduling relations, noted 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.

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

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.

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

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

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

In a sampling equation $x = y$ 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 $y \rightarrow^{\hat{x}} x$.

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

In a merge equation $x = y$ 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 $z \rightarrow^{\hat{z} \backslash \hat{y}} x$.

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

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 signal 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.

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

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

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

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

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

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 synchronization class of buffer. Two other synchronization 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.

$$R_{\text{buffer}} \stackrel{\text{def}}{=} \begin{pmatrix} \hat{x} = [t] \,|\, \hat{y} = [\neg t] \,|\, \hat{r} = \hat{x} \vee \hat{y} \\ s \rightarrow^{\hat{s}} t \,|\, y \rightarrow^{\hat{y}} r \;\;|\, r \rightarrow^{\hat{x}} x \end{pmatrix} /rst$$

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 $R_{\text{buffer}} \models (\hat{r} = \hat{t})$.

3.3 Clock hierarchy

The internal data-structures manipulated by the Signal compiler for program analysis and code generation consist 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 $\preceq$. It is defined by inductive application of the following rules :

(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 $R \models b_1 = c_1 f c_2$, $f \in \{\wedge, \vee, \backslash\}$, $b_2 \preceq c_1$, $b_2 \preceq c_2$ and b_2 is maximal (in the sense that $b_2 \succeq b$ for any b such that $b \preceq c_1$ and $b \preceq c_2$) 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.

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 $R \models b_1 = c_1\, f\, c_2$, $f \in \{\wedge, \vee, \setminus\}$, $b_2 \preceq c_1$, $b_2 \preceq c_2$ and b_2 is maximal then $b_2 \preceq b_1$.

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

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 $\{\hat{y}, [\neg t]\}$.

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

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

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

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 next.

3.4   Disjunctive form

But, before that, Polychrony 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

$$y \rightarrow^{[\neg t]} r \; {}^{[t]} \leftarrow r \text{ pre false}$$

In [**?**], 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 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 possibly instead.

Example In the case of process **current** in the example of the buffer one has that

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

Hence $\hat{x} \succeq \hat{t} \preceq \hat{y}$ and therefore $\hat{r} \setminus \hat{y}$ can be interpreted as $[t]$.

Timing relations are in disjunctive form iff they has no clock defined by a symmetric difference relation. For instance, suppose that $d \sim [x]$ and that $c \succeq b \preceq d$. Then, the expression $c \setminus d$ 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.*
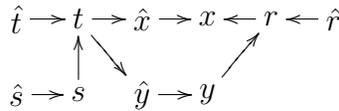
3.5 Scheduling graph

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 $\hat{y}$.

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

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. $S \models \hat{x} \to^{\hat{x}} x$ 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 $S \models y \to^{\hat{y}} \hat{x}$. 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 $S \models \hat{y} \to^{\hat{y}} \hat{x} \,|\, \hat{z} \to^{\hat{z}} \hat{x}$. 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 lighten the depiction). Notice that t is now scheduled before the clocks $\hat{x}$ and $\hat{y}$.

$$\hat{t} \twoheadrightarrow t \twoheadrightarrow \hat{x} \twoheadrightarrow x \twoheadleftarrow r \twoheadleftarrow \hat{r}$$
$$\hat{s} \twoheadrightarrow s \qquad \hat{y} \twoheadrightarrow y$$

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 $R \models a \to^c b$ then $R \models a \twoheadrightarrow^c b$. This just tells that the construction of the transitive closure relation $\twoheadrightarrow$ starts from the scheduling graph $\to$ of the process.

2. if $R \models a \twoheadrightarrow^c b$ and $R \models a \twoheadrightarrow^d b$ then $R \models a \twoheadrightarrow^{c \vee d} b$. If b is scheduled after a at clock c and at clock d then so it is at clock $c \vee d$

3. if $R \models a \twoheadrightarrow^c b$ and $R \models b \twoheadrightarrow^d z$ then $R \models a \twoheadrightarrow^{c \wedge d} z$. 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 $R \models e = 0$.*

3.6 Sequential code generation

Together with the control-flow graph implied by the timing relations of a process, the scheduling graph is used by Polychrony 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.

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

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.

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 $t \to y \to x$ 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.

3.7   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.

$$\hat{r} \sim \hat{s} \sim \hat{t}$$
$$[t] \sim \hat{x} \qquad\qquad [\neg t] \sim \hat{y}$$
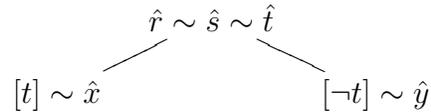
Formally, whatever the exact time samples t_1 and t_2 at which it receives an input signal y, or the time samples u_1 and u_2 at which it sends an output signal x, the buffer always behaves according to the same timing relations: t_i occurs strictly before u_i and s is always used at t_i and u_i.

		.	.	.	.		.	.	.	.	.	.	.
y		t_1		t_2			t'_1			t'_2			
s		t_1	u_1	t_2	u_2		t'_1		u'_1	t'_2		u'_2	
x			u_1		u_2				u'_1			u'_2	
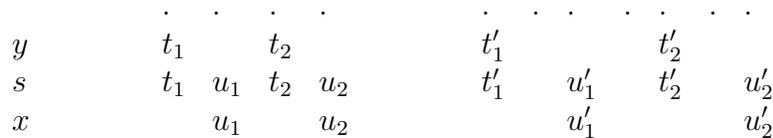
The timing relations between the signals x and y of the buffer are independent from latency incurred by communications with its environment: this is the formal definition of endochrony given in [9].

4   Compositional design criterion

We shall revisit the above schema in light of the compositional design methodology to be presented. We start by formulating a decision procedure that uses the clock hierarchy and the scheduling graph of a Signal process to compositionally check the property of isochrony.

Compilability   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 acyclic and its relations R are well-clocked.*

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

Proof   An immediate consequence of Property 5, in [20], where a well-clocked and acyclic process is proved to be deterministic.

Roots of a hierarchy   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 $\preceq$.

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

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

Definition 11 *A process p is* hierarchic *iff its hierarchy has a unique root.*

Property 2 *A compilable and hierarchic process p is endochronous.*

Proof   A detailed proof appears in [20].

Example   The hierarchies of process filter (Section 1), left, and of the buffer, right, are both hierarchic: they are endochronous. Let $e = ([y] \wedge [\neg z]) \vee ([\neg y] \wedge [z])$ and $f = ([\neg y] \wedge [\neg z]) \vee ([y] \wedge [z])$,
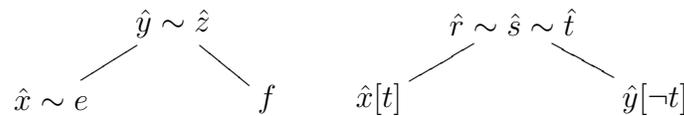
By contrast, a process with several roots necessarily defines concurrent threads of execution. Indeed, and by definition of a hierarchy, its roots cannot be expressed or calculated (or, a fortiori, synchronized or sampled) one with the others. Hence, they naturally define the source of concurrency for 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 by using bounded model checking. Property (2a) can be formulated as an invariant in Signal and submitted to its model checker Sigali [10].

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

The invariant returned by $\text{StateIndependent}\,(x,y)$ is defined for all pairs of root clock equivalence classes. 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 $(\neg cx_t \vee cy_t) \vee (cx_{t+1} \vee \neg cy_{t+1}) \vee (cx_t \wedge cy_t)$.

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.

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

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

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

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

Unfortunately, model-checking is unaffordable for purposes such as program transformation or code generation. In the aim of 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 hierarchic then it is* weakly hierachic. *If P and Q are weakly hierarchic, $P\,|\,Q$ is well-clocked and acyclic then $P\,|\,Q$ is weakly hierarchic.*

By induction on its structure, a process P is weakly hierarchic 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 hierarchic and the pair $(\prod_{j=1}^{i} P_j, P_{i+1})$ is well-clocked and acyclic.

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

Proof

1. By definition, a weakly hierarchic process P consists in the composition of a series of processes P_i that are individually compilable and hierarchic, 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 hierarchic) and at a clock c_j, computed using r_j (since P_j is hierarchic).

$$r_i \qquad\qquad r_j$$
$$\{c_i, \hat{x}, c_j\}$$
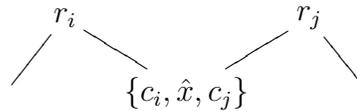
Since $P_i\,\|\,P_j$ is well-clocked, the clocks c_i, c_j and hence $\hat{x}$ have a disjunctive form. Hence, it cannot be the case that $\hat{x}$ is defined by the symmetric difference of a clock under r_i and another (e.g. under r_j). Therefore, any reaction initiated in P_i to produce $\hat{x}$ can 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 [18], namely, that non-blocking composition (2) of weakly endochronous processes (1) is isochronous. Consequently, the composition of P and Q is isochronous.

A compositional design methodology Our static criterion for checking the composition of endochronous processes isochronous defines a cost-effective methodology for the integration of components 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. [11]).

Example of a loosely time-triggered architecture We consider a simple yet realistic case study build upon the examples we previously presented. We wish to design a simulation model for a loosely time-triggered architecture (LTTA). 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.
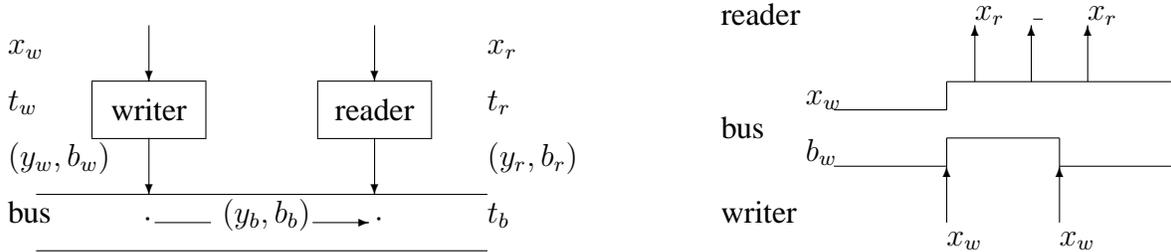
A simulation model of the LTTA To model an LTT architecture in Signal, we consider two data-processing functions that communicate by writing and reading values on an LTT bus. In Signal, we model an interface of these functions that exposes their (limited) control. The writer accepts an input x_w and 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.

$$x_r = \mathsf{reader}(y_r, b_r, c_r) \stackrel{\text{def}}{=} (x_r = y_r \,\mathsf{when}\,\mathsf{filter}(b_r) \, | \, \hat{y}_r = [c_r])$$
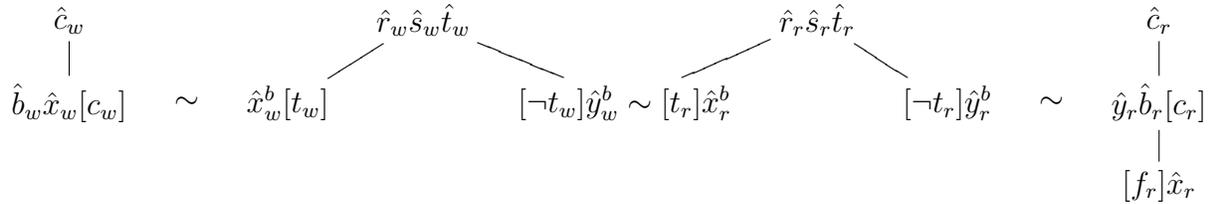
The bus buffers and forwards the inputs y_w and b_w to the reader. The clock c_b is not used since the buffers have local clocks.

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

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

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

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.

$$\begin{array}{cccccc}
\hat{c}_w & & \hat{r}_w \hat{s}_w \hat{t}_w & & \hat{r}_r \hat{s}_r \hat{t}_r & \hat{c}_r \\
\mid & & \diagup \qquad \diagdown & & \diagup \qquad \diagdown & \mid \\
\hat{b}_w \hat{x}_w [c_w] & \sim & \hat{x}_w^b [t_w] & [\neg t_w]\hat{y}_w^b \sim [t_r]\hat{x}_r^b & [\neg t_r]\hat{y}_r^b & \sim & \hat{y}_r \hat{b}_r [c_r] \\
& & & & & \mid \\
& & & & & [f_r]\hat{x}_r
\end{array}$$

5 A compositional code generation scheme

The above design methodology invites us to revisit the code generation process of Polychrony in the aim of implementing a separate compilation technique which accommodates the concurrent composition of endochronous processes by synthesizing rendez-vous protocols to compositionally interface processes. As we observe, it defines a new way to regard design using a synchronous multi-clocked model of computation by the component-based integration of endochronous functionalities, hence favoring modular exploration of the design space. We start this exposition by a careful analysis on the current features and limitations of Polychrony's code generator.

5.1 Current scheme

Both sequential, concurrent and distributed code generation schemes in Polychrony rely on the property of endochrony to generate the code. This observation also holds for code generation in related synchronous languages, Lustre and Esterel, without much salient difference. However, it is well-known that endochrony is not preserved by composition. To illustrate that, consider the following pair of processes, a producer and a consumer. The producer increments its output u when its input a is true and increments its output x otherwise.

$$(u, x) = \mathsf{producer}(a) \overset{\mathrm{def}}{=} \left(\begin{array}{ll} \hat{u} = [a] & \| u = 1 + (u \,\mathsf{pre}\, 0) \\ \| \hat{x} = [\neg a] & \| x = 1 + (x \,\mathsf{pre}\, 0) \end{array} \right) \qquad \begin{array}{c} \hat{a} \\ \diagup \quad \diagdown \\ [a] \sim \hat{u} \quad [\neg a] \sim \hat{x} \end{array}$$
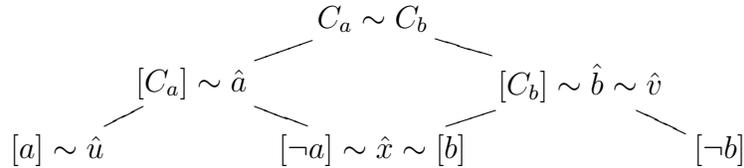
The consumer adds the value of x to the count v when b is true and 1 otherwise. Hierarchies are depicted on the right.

$$y = \mathsf{consumer}(b, x) \stackrel{\mathrm{def}}{=} \left(\begin{array}{rcl} \hat{v} & = & \hat{b} \\ | & \hat{x} & = & \mathsf{when}\, b \\ | & v & = & (v\,\mathsf{pre}\,0) + (x\,\mathsf{default}\,1) \end{array} \right)$$

$$\begin{array}{c} \hat{b} \sim \hat{v} \\ \diagup \quad \diagdown \\ [b] \sim \hat{x} \qquad [\neg b] \end{array}$$

The signal $x\,\mathsf{default}\,1$ is implicitly created. It has the same clock as b, its value is that of x at the clock $[b]$ and 1 at the clock $[\neg b]$. Notice that the producer and the consumer are endochronous (their hierarchies are trees). Now, consider the composition of producer and consumer in the main process below.

$$(u, v) = \mathsf{main}(a, b) \stackrel{\mathrm{def}}{=} \left(\begin{array}{rcl} (u, x) & = & \mathsf{producer}(a) \\ | & v & = & \mathsf{consumer}(b, x) \end{array} \right)$$

Polychrony produces a hierarchy in which two synchronized boolean signals C_a and C_b are added on top of the hierarchies of the producer and the consumer. This allows to (artificially) form an endochronous simulation process that relies on the environment to determine when to read a and/or b.

$$\begin{array}{c} C_a \sim C_b \\ \diagup \qquad \diagdown \\ [C_a] \sim \hat{a} \qquad\qquad [C_b] \sim \hat{b} \sim \hat{v} \\ \diagup \qquad \diagdown \qquad\qquad \diagdown \\ [a] \sim \hat{u} \qquad [\neg a] \sim \hat{x} \sim [b] \qquad\qquad [\neg b] \end{array}$$

This structure yields the generation of code that differs from what we have seen so far in that the transition function now expects the clocks C_a and C_b to be synchronously delivered by the environment, instead of being computed internally.

```
bool main_iterate() {                                      }
    if (!r_main_C_a(&C_a)) return FALSE;               }
    if (!r_main_C_b(&C_b)) return FALSE;           C__63 = (C_a ? C_ : FALSE);
    if (C_b) {                                     if ((C_) != b)
        if (!r_main_b(&b)) return FALSE;               polychrony_exception
    }                                                      ("Exception for (C_, b)");
    if (C_a) {                                     if (C_b) {
        if (!r_main_a(&a)) return FALSE;               if (C__63) XZX_36 = x;
        C_ = !a;                                       else XZX_36 = 1;
        if (a) {                                       v = v + XZX_36;
            u = 1 + u;                                 w_main_v(v);
            w_main_u(u);                           }
        }                                          C_ = FALSE;
        if (C_) {                                  return TRUE;
            x = 1 + x;                         }
```

In the C code, the functions r_main_C_a, r_main_C_b, r_main_a and r_main_b read the input signals C_a, C_b, a, b and the functions w_main_u and w_main_v write the outputs u and v.

The compiler places the clocks $[\neg a]$, $\hat{x}$ and $[b]$ in the same equivalence class. However, one easily notices that the equation $[\neg a] = [b]$, incurred by the composition of the producer and the consumer, is non-trivial. At present, it is bailed out as a so-called "clock constraint" by Polychrony.

To handle this clock constraint, Polychrony can either generate a proof obligation, which will have to be checked by the user, or generate defensive code to raise an exception if the clock constraint is violated during execution. In the generated code below, if `!a != b`, an exception is reported by the simulation loop.

The functionality of Polychrony to detect and report such a constraint is central in the code generation scheme that will be presented next. Let us have a second look at the present situation:

- the producer and the consumer are endochronous
- the signal x is defined in one process, the producer
- its clock $\hat{x}$ is used in both processes

In the composition of the consumer and the producer, one can hence define x by a shared variable and use its clock constraint to define when it can deterministically be defined and/or used by either the processes.

5.2 Contributed code generation scheme

Our contribution builds upon this simple idea, that is suitable for the simulation of otherwise deterministic specifications, and uses the facility of Polychrony to report clock constraints (such as $[b] = [\neg a]$) and to export independent clocks (such as C_a and C_b) to build a scheduler that satisfies the expected safety properties.

In this aim, and first of all, we would like to avoid increasing the interface of the program (with C_a or C_b) in order to have an efficient (sequential or concurrent) execution scheme. In the present code generation scheme of Polychrony, C_a and C_b are added to rebuild an endochronous simulation loop.

In the present case, however, the composition of the producer and the consumer is weakly endochronous: the very interleaving of a and b during execution is not relevant to the correct propagation of input and output values. The transitions involving only a or only b may be executed in any order. However, transitions involving both a and b need to be synchronized. This is precisely where the clock constraint $[b] = [\neg a]$ comes into play.

Building a controller

Using the information provided by Polychrony, namely, the exportation of non-hierarchized clocks C_a and C_b, the report of a clock constraint on shared signals such as $[b] = [\neg a]$, we can easily build a process for controlling the execution of the composition of the producer and the consumer so as to keep it within a suitable safety objective.

To allow for a correct resynchronization on the values of x, the controller needs to obey the requirement expressed by the clock constraint $[\neg a] = [b]$ while imposing no additional synchronization constraint (on a or b).

Nicely, this controller can be expressed and synthesized in Signal. It uses the clock constraint of the shared variable ($\hat{x} = $ when not $a = $ when b) to synchronize instants that need to be.

The controller accepts the input signals a and b and feeds the producer and the consumer with copies c and d until one of the constraints is met, [when not a] or [when b]. As soon as this occurs, it stops reading input from the signal (a or b), suspending the corresponding process, until the other meets the constraint.

$$(c, d) = \mathsf{controller}(a, b) \stackrel{\mathrm{def}}{=} \left(\begin{array}{l} c = \mathsf{scheduler}(a, r_a, r) \\ \mid \ d = \mathsf{scheduler}(b, r_b, r) \\ \mid r_a = \ \mathsf{not}\, a\, \mathsf{default}\, (r_a\, \mathsf{pre\, false}) \\ \mid r_b = b \quad\ \mathsf{default}\, (r_b\, \mathsf{pre\, false}) \\ \mid \ r = r_a\, \mathsf{and}\, r_b \end{array} \right) / r_a r_b r$$

The controller contains two schedulers that are responsible for suspending and resuming the input signals a and b (hence the producer and the consumer) in order to correctly schedule the operations in the sequential implementation of the rendez-vous.

$$y = \mathsf{scheduler}(x, r_x, r) \stackrel{\mathrm{def}}{=} \left(\begin{array}{l} \hat{x} = \ \mathsf{true\ when}\, c_x \\ \mid r_x = \ \mathsf{not}\, a\, \mathsf{default}\, r'_x \\ \mid r'_x = r_x\, \mathsf{pre\ false} \\ \mid c_x = \qquad\quad (\, \mathsf{true\ when}\, (r\, \mathsf{pre\ false})) \\ \qquad\quad \mathsf{default}\ (\, \mathsf{false\ when}\, r'_x) \\ \qquad\quad \mathsf{default}\ \ \mathsf{true} \\ \mid c_y = (c_x\, \mathsf{and\ not}\, r_x)\, \mathsf{or}\, r \\ \mid \ \ y = (x\, \mathsf{cell}\, c_y)\, \mathsf{when}\, c_y \end{array} \right) / c_x c_y$$

Last, we need to patch the main program with the controller to correctly feed the producer and the consumer with the values of a and b that satisfy the clock constraint.

$$(u, v) = \mathsf{main}(a, b) \stackrel{\mathrm{def}}{=} \left(\begin{array}{r} (u, x) = \mathsf{producer}(c) \\ \mid \quad\ v = \mathsf{consumer}(d, x) \\ \mid (c, d) = \mathsf{controller}(a, b) \end{array} \right) / cdx$$

Notice that each of the producer and the consumer is able to independently react when either [$\neg a$] or [b] holds, as no synchronization needs to take place in those cases.

Sequential code generation scheme

The controller is build upon the clocks exported and the constraints reported by Polychrony. This provides sufficient information to generate the necessary code to control the execution of the composition of endochronous processes.

In the controlled main program, variables prefixed with `pre_` register the values of signal (of corresponding suffix) until the next cycle. The generated `r` variables translate the synchronization obligation implied by the reported clock constraint as `r = ra && rb`. Functions named `{r|w}_main_x` read and write the signal x.

As opposed to the generated code presented Section **??**, the present program does not need its master clocks to be synchronized: C_a and C_b are local variables, not input signals. As a result, the interface of the composition of the producer and the consumer is the union of interfaces.

We observe that, since the producer and the consumer are endochronous, and since their composition is such that all clocks which can be computed (all have a disjunctive form), the main program is weakly isochronous in the sense of [17]: any synchronous reaction, initiated from one side, yields a globally isochronous execution. This yields to a generic methodological principle, presented next.

```
bool main_iterate() {                       /*  (x,u) = producer (c) */

    /* c = scheduler (a, ra, r) */          C_1 = FALSE;
                                            if (C_c) {
    if (pre_r) C_a = TRUE;                      C_1 = !a;
       else if (pre_ra) C_a = FALSE;            if (a) {
       else C_a = TRUE;                             u = 1 + u;
    if (C_a) {                                      w_main_u(u);
        if (!r_main_a(&a)) return FALSE;        }
    }                                           if (C_1) x = 1 + x;
    if (C_a) ra = !a;                       }
       else ra = pre_ra;
                                            /*  y = consumer (d,x) */
    /* d = scheduler (b, rb, r) */
                                            C_2 = (C_c ? C_1 : FALSE);
    if (pre_r) C_b = TRUE;                  if (C_d) {
       else if (pre_rb) C_b = FALSE;            if (C_2) X_1 = x;
       else C_b = TRUE;                            else X_1 = 1;
    if (C_b) {                                  v = v + X_1;
        if (!r_main_b(&b)) return FALSE;        w_main_v(v);
    }                                       }
    if (C_b) rb = b;
       else rb = pre_rb;                    /* finalisation */

    /* main */                             pre_ra = ra;
                                           pre_rb = rb;
    r = ra && rb;                          pre_r = r;
    C_c = (C_a && !ra) || r;               return TRUE;
    C_d = (C_b && !rb) || r;           }
```

Compositionality

In our example, we observe that, should the main process be composed with an additional endochronous process (or weakly endochronous network), then we would only need to build an additional controller between those two, based on the same principle as previously mentionned: to capture the clocks exported by Polychrony and to implement rendez-vous between toplevel clock constraints (here: $\hat{b} = [c]$) in the hierarchy.

$$(u, w) = \mathsf{main2}(a, b, c) \stackrel{\mathrm{def}}{=} ((u, v) = \mathsf{main}(a, d) \,|\, w = \mathsf{consumer}(e, v) \,|\, (d, e) = \mathsf{controller2}(b, c)) \,/de$$

Concurrent code generation scheme

The generation of code for concurrent execution differs from sequential code generation by the construction of clusters that match the physical partition of signals on the target execution architecture. In the present case, these clusters are the composed endochronous processes, the producer and the consumer.

Our compilation technique for sequential code generation can easily be adapted for concurrent execution. It allows to define an interface or controller that performs minimum arbitration with its environment. As a result, producer and consumer are compiled separately and the global safety guarantee of weak isochrony is relied on assess the safety of the concurrent composition.

```
pthread_barrier_t *begin_RDV, *end_RDV ;
pthread_barrier_init(begin_RDV, 2);
pthread_barrier_init(end_RDV, 2);
```

In the example, we have separately compiled the producer and consumer to ready them for concurrent execution. They use the local read/write functions of the producer and the consumer: $\{r|w\}\_\{consumer|producer\}\_x$). The clock constraint $[\neg a] = b$ is again used to synchronize the threads with a barrier: a mutex zone RDV protects the shared variable x.

```
bool consumer() {                          bool producer() {
  if (!r_consumer_b(&b))                     if (!r_producer_a(&a))
      return FALSE;                               return FALSE;
  if (b) {                                   if (a) {
     pthread_barrier_wait(begin_RDV);           u = 1 + u;
     X_1 = x;                                    w_producer_u(u);
     pthread_barrier_wait(end_RDV);          }
  } else X_1 = 1;                            if (!a) {
  v = v + X_1;                                  pthread_barrier_wait(begin_RDV);
  w_consumer_v(v);                             x = 1 + x;
  return TRUE;                                  pthread_barrier_wait(end_RDV);
}                                            }
                                             return TRUE;
                                           }
```

The generated code is otherwise unchanged. We obtain a concurrent code generation scheme that modularly and compositionally supports separate compilation. It efficiently uses existing report functionalities of the present implementation of Polychrony to effectively support the synthesis of a controller that is able to assemble endochronous processes so as to maintain a global objective of weak isochrony.

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 adhoc 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 proposed

in [12]. In the case of Signal, automated distribution is proposed by Aubry [1]. It consists in partitioning endochronous specifications and synthesizing inter-partition protocols to ensure preservation of endochrony.

In [13], 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 [18], 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 [19], 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 [8], 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 [5], 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 [20]) 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 [7] 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 [7] is a "grey-box" approach, where knowledge on the pearl is needed to synthesize an an-hoc controller and, at the same time, ensure functional correctness.

7   Conclusions

The clock analysis at the core of our approach shares similarities with both approaches (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 [11]). As an example, TNI has developed a real-time, hardware in-the-loop, simulator of all 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] Pascal Aubry. Mises en oeuvre distribuées de programmes synchrones. Thèse de l'Université de Rennes, 1997.

[2] Loïc Besnard. Compilation de Signal: horloges, dépendances, environnements. Thèse de l'Université de Rennes, 1992.

[3] Albert Benveniste, Benoit Caillaud, and Paul Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, v. 163. Academic Press, 2000.

[4] 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.

[5] 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.

[6] Paul Caspi, Alain Girault, and Daniel Pilaud. Distributing Reactive Systems. International Conference on Parallel and Distributed Computing Systems. ISCA, 1994.

[7] Mario Casu, Luca Macchiarulo. A new approach to latency insensitive design. Design Automation Conference. ACM, 2004.

[8] Sohini Dasgupta, Dumitru Potop-Butucaru, Benoît Caillaud, Alex Yakovlev. Moving from Weakly Endochronous Systems to Delay-Insensitive Circuits. Formal Methods for GALS Design, Electronic Notes in Theoretica Computer Science. Elsevier, 2006.

[9] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*. World Scientific, 2003.

[10] 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.

[11] Abdoulaye Gamatié, Thierry Gautier. Synchronous Modeling of Avionics Applications using the SIGNAL Language. Real-Time and Embedded Technology and Applications Symposium. IEEE, 2003.

[12] 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.

[13] Alain Girault, Xavier Nicollin, and Marc Pouzet. Automatic rate desynchronization of embedded reactive programs. ACM Transactions on Embedded Computing Systems, 5(3). ACM, 2006.

[14] LEE, E., SANGIOVANNI-VINCENTELLI, A. "A framework for comparing models of computation". In *IEEE transactions on computer-aided design*, v. 17, n. 12. IEEE Press, December 1998.

[15] Olivier Maffeïs. Ordonnancements de graphes de flots synchrones ; application à la mise en oeuvre de SIGNAL. Thèse de l'Universit de Rennes, 1993.

[16] 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.

[17] Dimitru Potop-Butucaru and Benoit Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Application of concurrency to system design*. IEEE, 2005.

[18] Dimitru Potop-Butucaru, Benoit Caillaud, and Albert Benveniste. Concurrency in synchronous systems. In *Formal Methods in System Design*. Kluwer, 2006.

[19] 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.

[20] 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.

[21] 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, February 2008.

Appendix

The appendix recall the semantics of synchronization and scheduling relations in the polychronous model of computation, presented in [9]. It is complementary material for information to reviewers. A scheduling structure can be added to the polychronous model of computation outlined in the present article to define a denotational semantics of scheduling relations $x \rightarrow^c y$.

Scheduling structure   To render scheduling relations between events occurring at the same time tag t, we equip the domain of polychrony with a scheduling relation, noted $t_x \rightarrow t'_y$, defined on a domain of dates $\mathcal{D} = \mathcal{T} \times \mathcal{X}$, to mean that the event along the signal named y at t' 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 constraint 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

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

Meaning of clocks   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.

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

Meaning of scheduling relations   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 $t_y \rightarrow^b t_x$.

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

In [9], we finally show that, whenever a process P has graph R, then $[\![P]\!] \subseteq [\![R]\!]$.