The Polychronous Model of Computation and Kahn Process Networks

Thierry Gautier[a+], Paul Le Guernic[a+], Loïc Besnard[b+], Jean-Pierre Talpin[a*]

[a] *Inria,* [b] *CNRS,* [+] *ret., IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France.*

Abstract

In 1974, Gilles Kahn defined a seminal semantic model for asynchronous dataflow programming that would then be called as the eponymous Kahn process networks (KPN) and instantiated in as many models of the so-called DPN hierarchy as domain-specific fields of information processing from digital signal processing to hybrid cyber-physical systems. Among these, synchronous programming models have had an important impact in the specific domain of embedded system design. In this paper, we consider an instance of what seems to be one of the many synchronous models of computation: polychrony, initiated by the dataflow language Signal and its multi-clock (i.e. polychronous) model of computation and, later on, CCSL (clock constraints specification language). We provide an in-depth study of its semantic relationships with respect to the original definition of KPNs and hint toward the idea of polychrony as a methodology to locally synchronize (abstractions of) globally asynchronous processes. In particular, we formally define the property, referred to as "polyendochrony", that allow to consider a given desynchronized network of synchronous Signal processes (a GALS architecture) as an implementation of a corresponding KPN model (an asynchronous network of Khan-deterministic functions). For this class of networks, we formalize the Signal program analysis and transformations that defines synchronous clusters of Signal processes of guaranteed deterministic behavior in an asynchronous network, that is, without synchronizing communications in the entire network. This definition yields a new strategy of multi-threaded code generation that is available in the open-source Polychrony toolset of the Signal language and blurs the limits between asynchronous and polychronous models of computation.

Keywords: synchronous languages, semantics of programming languages, formal methods, synchrony and asynchrony, Kahn process networks

*Corresponding author

Email address: Jean-Pierre.Talpin@inria.fr (Jean-Pierre Talpin[a])

1. Introduction

Models of cooperating processes have a long history in both theory and practice of computer science. On the one hand, theory, process calculi describe interactions and communications between processes with algebraic laws that permit to express equivalence relations and to formally reason about process expressions. Well-known models are CSP [1], CCS [2], ACP [3], or the π-calculus [4], among many others. Other formal models of computation and communication (MoCCs) provide a central notion of process, with sometimes a real proximity with effective industrial practice. This is the case for instance of Dataflow Process Networks [5], which are used in particular in signal processing software. Dataflow Process Networks are themselves a special class of Kahn Process Networks [6], which are one of the origins of the dataflow concept. The dataflow model handles general parallel asynchronous execution of processes.

On the other hand, practice, the design of embedded systems, and more specifically safety critical systems, has given rise to models based on the so-called "synchronous hypothesis" (which roughly consists in abstracting non-functional implementation details, essentially real-time, from a system under design) and to associated "synchronous languages" [7, 8]. In the class of synchronous languages, or close to it, the refinement-based Signal specification language [9, 10, 11], based on the "polychronous MoCC", is characterized by its ability to describe systems with multiple clocks and verifiably transform them to globally asynchronous, locally synchronous systems, prior to multi-threaded, multi-core, dynamic or static real-time scheduling analysis. The language adopts a dataflow style, but flows of data are *synchronized* through these clocks.

Traditionally, models of cooperating processes are classified along two aspects, which are *cooperation* (or composition) and *communication*. These aspects are qualified using two criterias: *asynchronous* and *synchronous*. In this way, Bergstra *et al.* [12] situate CSP, CCS and ACP in the regime of asynchronous cooperation, while SCCS [13] follows synchronous cooperation; dataflow networks are considered as representative of asynchronous communication, while CSP, CCS or ACP obey to synchronous communication.

Considering the polychronous MoCC and the *synchronized dataflow* [14] language Signal, this classification based on the distinction *synchronous/asynchronous* could be refined using a third criteria which is precisely "synchronized". Thus, regarding *composition* (cooperation), we distinguish:

- *asynchronous*: processes are driven by different non-synchronized clocks, each one progressing at its own pace;
- *synchronized*: some implicit or explicit synchronization rules are introduced (relations on communications);
- *synchronous*: processes are driven by a single global clock such that an "atomic" action is activated at each tick.

With this classification, CCS (for instance) would be considered as following synchronized composition rather than asynchronous one. Regarding *communication* (interaction) we also distinguish:

- *asynchronous*: asynchronous interaction indicates the fact that data transfer between processes may take an *unbounded* amount of time and may need an *unbounded* amount of memory to hold sent and unreceived values;
- *synchronized*: some implicit or explicit synchronization rules are introduced (bounded FIFOs may be used);
- *synchronous*: communication occurs in a fixed number of ticks of each participant (rendez-vous...).

In this paper, we provide an in-depth overview of the relation between two models of processes: the one of asynchronous (or untimed) dataflow represented by Kahn Process Networks, and that of synchronized dataflow represented by the polychronous model. In particular, the fact that processes in the polychronous model (represented by Signal programs) can be asynchronously nondeterministic has to be carefully and analytically taken into account.

In general, the relationship synchrony/asynchrony is crucial since on the one hand, synchronous programs may have to be at least partly desynchronized to be implemented efficiently onto possibly asynchronous architectures, and on the other hand, asynchronous programs need to be synchronized in order to get effective, correct-by-construction, implementations.

Previous works

In this aim, a number of theoretical results have been formulated, and reference papers on the semantic model, including formal properties for compositionality and asynchronous implementation, consider this issue more or less explicitly under the term of isochrony.

Isochrony is the property of (at least) two processes to indifferently produce the same flows of values, whether synchronously or asynchronously composed. Isochrony is a much desirable property, as it implies the capability to design and schedule processes compositionally, that is, independently of their runtime scheduler or architecture. A ground motivation for seeking isochrony is *desynchronization*: the transformation of a monolithic synchronous design into several tasks to be executed in parallel.

The first formal address of *desynchronization* can be found in [15], where precise relations between well-clocked synchronous functional programs and the subset of Kahn-networks are established, and shown to be amenable to bufferless evaluation. In [16], the author considers the distribution of synchronous automata on asynchronous networks using FIFO-buffered broadcast communications. In [17], a model for the distribution of synchronous programs on distributed architectures is introduced which uses low-level non-blocking one-place buffers.

In [18], an analysis of the links between synchrony and asynchrony is presented in the context of synchronous transition systems and the compositional property of isochrony introduced (a pair of processes is isochronous iff its synchronous and asynchronous compositions admit the same traces). It provides a first intuition for defining the property of isochrony from that of endochrony: the capacity of an individual process to determine the status (present or absent) of all events, at all times.

As this property is however too strong (endochronous processes are not isochronous, in general), a number of subsequent works refined it by weakening it to endo-isochrony [10, 19], by first repurposing the static analyser for endochrony, and then by defining the the largest class of such isochronous processes under the term of weak endochrony [20, 21, 22]. In turn, weak endochrony is the capacity of a process to always make its environment aware when it locally choose an event to be absent (relevant absence). It does, in particular, correspond to the common practice of flow-control in the design of asynchronous protocols.

Isochrony also relates to the notion of latency-insensitive circuits of Carloni et al. [23], which we formally characterize and compare with in Section 2.7.

Goal

In this paper, we revisit the above line of works by an analysis of the polychronous model of computation of Signal under the perspective of its semantic relations with the original model of Kahn process networks, knowing that both models have given rise to efficient implementations in different domains of application such as time-critical systems and stream processing. We will make the choice to present and expand these formal results by considering them under a close relationship to the present implementation of the Signal language in the open-source Polychrony toolset[1], which does not integrate advanced analysis to synthesize weakly-endochornous networks.

Plan

The rest of this article is organized as follows. In Section 2, we review the formal basis of the polychronous model and introduce characteristic properties and categories of processes. In particular, we define the class of so-called polyendochronous processes [24]. Then we provide typical simple Signal specifications. In Section 3, we first review the definition of KPNs as flow functions. Then we define Signal operators as extended flow functions and show in which conditions Signal programs can be seen as KPNs. In this context, we consider different cases of process networks, including polyendochronous processes. Then, in Section 4, we describe code generation for the class of such polyendochronous processes. Finally, in Section 5, we propose an extension to Signal allowing to consider (bounded) KPNs as (extended) Signal programs.

[1] `http://polychrony.inria.fr`

2. Signal polychronous model

Several semantics of the Signal language have been provided along years. Our presentation of the polychronous model is mainly based on [10], with some revisions. Basics of the model are defined in Section 2.1, including synchronous composition of processes. Then, Sections 2.2 and 2.3 provide different order and equivalence relations allowing to compare processes. The asynchronous composition of processes is defined in 2.4, and the property of flow-invariance allows to relate synchronous and asynchronous compositions. A special category of "endochronous processes" is defined in Section 2.5, deterministic processes are characterized in Section 2.6, and weakly endochronous processes (with, among them, polyendochronous processes) are defined in Section 2.7. Then, the semantics of Signal basic operators is described in Section 2.8 and useful derived operators are defined in 2.9. The use of these operators is illustrated in a few typical examples in Section 2.10.

2.1. Basics of the polychronous model

In the polychronous model, a component will be abstracted by its time relations on its observable variables.

Time domain. The following requirements are considered for a *time domain* (or *tag domain* since it is not necessarily directly associated with a numeric notion of time) Ω:

- Ω is a poset equipped with a partial order relation $\leq$.
- Ω is a meet-semilattice: any two elements in Ω have an infimum (or greatest lower bound) in Ω. This allows for time causality.
- Ω is a densely ordered set: for all x and y in Ω for which $x \leq y$, there exists z in Ω such that $x < z < y$. This allows for context adaptation and implementation refinement.
- Any finite family of subsets of Ω has an upper bound in Ω.

A chain C of Ω is a countable and well-ordered subset of Ω. Such a chain C is well-ordered if all its non-empty subsets have a minimal element for $\leq$. In particular, a (countable and well-ordered) chain C has a lower-bound for $\leq$, written $\min(C)$ [10].

We denote by $\mathcal{C}$ the set of such chains C. For a nonempty chain $C \in \mathcal{C} \setminus \emptyset$, we write $\min(C)$ for the least element of C, and for a tag $t \in C \setminus \min(C)$, we write $\text{pred}_C(t)$ for the (unique) immediate predecessor of t in C.

Domain of values. The domain of values $\mathbb{D}$ is a set of values containing at least Boolean values ($\mathbb{B} = \{ff, tt\} \subseteq \mathbb{D}$).

Timed history.

Definition 1 (Timed history—a.k.a. signal—, event). A *timed history* (or a *signal*) is a mapping s from a chain $C \in \mathcal{C}$ to the domain of values $\mathbb{D}$ ($s : C \to \mathbb{D}$). We write $tags(s)$ to denote the domain of s. The pairs (t, v) in s such that $s(t) = v$ are called *events*.

Let $\mathcal{S}$ be the set of timed histories.

Behavior. We consider a countable set $\mathcal{V}$ of variables.

Definition 2 (Behavior). A *behavior* on V, where V is a finite set of variables of $\mathcal{V}$ ($V \subseteq \mathcal{V}$), is a mapping b from V to the set of timed histories ($b : V \to \mathcal{S}$).

For a variable $x \in V$, $b(x)$ is the signal (timed history) designated by x in b. We write $vars(b)$ to denote the domain of b and $tags(b)$ to denote its tags (or tag support): $tags(b) = \bigcup_{x \in vars(b)} tags(b(x))$. The relation $\leq_{tags(b)}$ on $tags(b)$ is the partial order relation defined by the union of order relations on chains of tags of the signals of b. The behavior on the empty set of variables is denoted by $\emptyset$. For any behavior b on V, we write $b_{|V'}$ for the projection of b on a set of variables $V' \subseteq V$. We write $\mathcal{B}_{|V}$ for the set of behaviors on variables V.

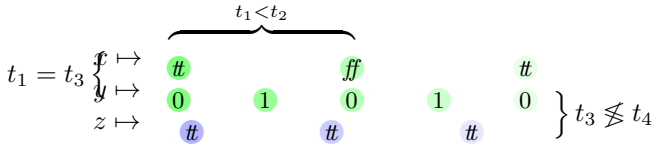


Figure 1: Depiction of a behavior b from signal names x, y, z to partially ordered tags and values. The 1^{st} tag t_1 of x precedes its 2^{nd} ($t_1 < t_2$) and equals the 1^{st} tag of y ($t_1 = t_3$) while z is synchronized with neither of x or y ($_3 \not\leqslant t_4$).

A behavior prefix (which is a finite behavior), written $b_{\leq \Theta}$, is the projection of the behavior b on the set of the tags $t \in tags(b)$ such that $\forall \theta \in \Theta, t \leq \theta$. Similar notations are considered for a behavior prefix until a tag θ and before a tag θ (respectively, $b_{\leq \theta}$ and $b_{<\Theta}$).

Concatenation of reactions. A *reaction* is a behavior with (at most) one tag. As for behaviors, we write $vars(r)$ to denote the domain of a reaction r and $tags(r)$ to denote the tag of a non empty reaction r.

A reaction r is concatenable to a behavior b iff $vars(b) = vars(r)$ and $\forall x \in vars(b)$, either $r(x) = \emptyset$ (where $\emptyset$ is the empty set of timed histories), or, if t is the tag of r, $t \leq u \Rightarrow u \notin tags(b(x))$. In that case, concatenating r to b is defined $\forall x \in vars(b)$, $\forall t \in tags(b) \cup tags(r)$, by $(b \cdot r)(x)(t) = r(x)(t)$ if $t \in tags(r(x))$ and $(b \cdot r)(x)(t) = b(x)(t)$ otherwise. In particular, reactions can be concatenated to form behaviors.

Process.

Definition 3 (Process). A *process* P on a finite set of variables $V \subseteq \mathcal{V}$ is a set of behaviors on V (V is also noted $vars(P)$).

There is a single process on the empty set of variables, which is $\{\emptyset\}$.

Let $\mathcal{P}$ be the set of processes.

The restriction $P_{|V'}$ of a process P on V, with $V' \subseteq V$, is defined by the set of behaviors $b_{|V'}$ such that $b \in P$.

Composition.

Definition 4 (Synchronous composition). The *composition* (or *synchronous composition*) $P_1|P_2$ of two processes P_1 and P_2, respectively on V_1 and V_2, is the process on $V_1 \cup V_2$ defined as the set of behaviors b such that $b_{|V_1} \in P_1$ and $b_{|V_2} \in P_2$.

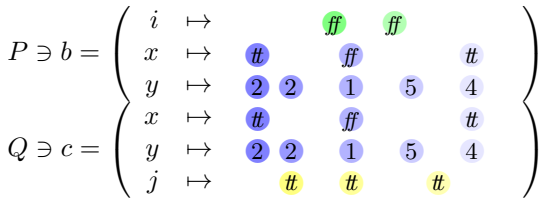


Figure 2: Synchronous composition of two behaviors b, c forming $P|Q$: signals common to P and Q (x, y) must hold the same tags and values whereas others (i, j) need not

Synchronous composition defines an "intersection" of behaviors and implies a syntactic congruence property:

- $(\mathcal{P}, |, \{\emptyset\})$ is a commutative monoid.
- The composition operator $|$ is idempotent.

2.2. Stretching and stretch-equivalence

A "stretching" relation is defined as follows on behaviors:

Definition 5 (Stretching). Let b_1, b_2 be two behaviors, b_2 is a *stretching* of b_1, written $b_1 \leq b_2$, iff $vars(b_1) = vars(b_2)$ and there exists a bijection $f : tags(b_1) \to tags(b_2)$ such that:
$\forall C \in \mathcal{C}, \forall t \in C, t \leq f(t)$,
$\forall t_1, t_2 \in tags(b_1), t_1 \leq t_2 \Leftrightarrow f(t_1) \leq f(t_2)$ (f is increasing),
and b_1 and b_2 are isomorphic:
$\forall x \in vars(b_1), f(tags(b_1(x))) = tags(b_2(x))$,
$\forall x \in vars(b_1), \forall t \in tags(b_1(x)), b_1(x)(t) = b_2(x)(f(t))$.

The stretching relation is a partial order on behaviors. An equivalence relation, called "stretch-equivalence", is defined from this partial order:

Definition 6 (Stretch-equivalence). Two behaviors b_1 and b_2 are *stretch-equivalent*, written $b_1 \gtrless b_2$, iff there exists a behavior c such that $c \leq b_1$ and $c \leq b_2$.

Intuitively, behaviors remain equivalent with a stretching or a translation in the time domain. Stretch-equivalence preserves the simultaneity and the ordering of events within a behavior. It is sometimes called clock-equivalence.

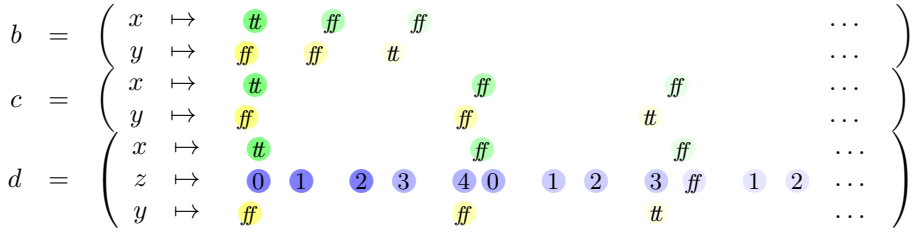


Figure 3: A depiction of stretching the time tags of behavior b as the behavior c, to allow for a scalable refinement of b as d and partial synchronization to a contextual "hyper-period" (z)

The equivalence class of a behavior for the stretch-equivalence forms a meet-semilattice which has a least element. We call *strict behaviors* the behaviors which are minimal for the stretching relation. For a given behavior b, the set of all behaviors that are stretch-equivalent to b defines its *stretch-closure*, written b^*. The stretch-closure of a set of behaviors p is the set denoted by p^*, which contains all the behaviors resulting from the stretch-closure of each behavior $b \in p$, i.e., $p^* = \bigcup_{b \in p} b^*$.

It is interesting, in particular, to consider the so-called *stretch-closed processes*, which are those processes p such that $p = p^*$. A non-empty stretch-closed process p admits a set of strict behaviors, written $(p)_{\gtrless}$, such that $(p)_{\gtrless} \subseteq p$ (for all $b \in p$, there exists a unique $c \in (p)_{\gtrless}$ such that $c \gtrless b$).

2.3. Relaxation and flow-equivalence

All stretch-equivalent behaviors have the same synchronization relations between their events. A weaker relation than stretching may be considered, which discards these synchronization relations and allows for comparing behaviors with respect to the sequences of values that variables hold.

Definition 7 (Relaxation). A behavior b_2 on V is a *relaxation* of a behavior b_1 on V, written $b_1 \sqsubseteq b_2$, iff for all $x \in V$, $b_{1|\{x\}} \leq b_{2|\{x\}}$ (the projection of b_2 on each variable is a stretching of the projection of b_1 on this variable).

The relaxation relation is a partial order on behaviors. An equivalence relation, called "flow-equivalence", is defined from this partial order:

Definition 8 (Flow-equivalence). Two behaviors b_1 and b_2 are *flow-equivalent*, written $b_1 \approx b_2$, iff there exists a behavior c such that $c \sqsubseteq b_1$ and $c \sqsubseteq b_2$.

The equivalence class of a behavior b for the flow-equivalence forms a meet semi-lattice which has a least element; it is a strict behavior, written $b_{\approx}$.

The following properties, which are easy to establish, will be used in some proofs:

- If $b \approx c$ and $X \subseteq vars(b)$ then $b_{|X} \approx c_{|X}$;
- If $b_{|X} \approx c_{|X}$ and $b_{|Y} \approx c_{|Y}$ then $b_{|X \cup Y} \approx c_{|X \cup Y}$.

2.4. Asynchronous composition and flow-invariance

The asynchronous composition (or parallel composition) of processes is more flexible than the synchronous composition. It allows behaviors that would be rejected in synchronous composition. The *asynchronous composition* of two processes P and Q contains all the behaviors b that are relaxations of behaviors b_1 and b_2 from P and Q along shared variables, and that are stretch-equivalent to behaviors b_1 and b_2 along independent variables of P and Q.

Definition 9 (Asynchronous composition). If processes P and Q are respectively defined on sets of variables V_1 and V_2, their *asynchronous composition* is:

$$P \mid\mid Q = \left\{ b \in \mathcal{B}_{|V_1 \cup V_2} \;\middle|\; \begin{array}{l} \exists b_1 \in P, b_{1|V_1 \cap V_2} \sqsubseteq b_{|V_1 \cap V_2} \wedge b_{1|V_1 \setminus V_2} \gtrless b_{|V_1 \setminus V_2} \\ \exists b_2 \in Q, b_{2|V_1 \cap V_2} \sqsubseteq b_{|V_1 \cap V_2} \wedge b_{2|V_2 \setminus V_1} \gtrless b_{|V_2 \setminus V_1} \end{array} \right\}$$

Flow-invariance, based on flow-equivalence, allows to relate synchronous and asynchronous compositions of processes.

Definition 10 (Flow-invariance). A process P defined on V is *flow-invariant* on $I \subseteq V$ iff
$\forall b_1, b_2 \in P, b_{1|I} \approx b_{2|I} \Rightarrow b_1 \approx b_2$
or, equivalently, $\forall b_1, b_2 \in P, (b_{1|I})_{\approx} = (b_{2|I})_{\approx} \Rightarrow b_1 \approx b_2$.

In particular, if P is the asynchronous composition of two processes, $P = p \mid\mid q$, the flow-invariance of P on some variables I implies the flow-equivalence between behaviors of $p \mid\mid q$ and behaviors of $p \mid q$ (since P contains all these behaviors). Thus, flow-invariance ensures that the refinement of a synchronous specification $p \mid q$ by an asynchronous implementation $p \mid\mid q$ preserves the sequences of values along all signals for any given behavior. Moreover, flow-invariance is compositional: if p and q are respectively flow-invariant on $I \cap vars(p)$ and $I \cap vars(q)$, then $p \mid\mid q$ is flow-invariant on I.

2.5. Endochronous processes

Endochronous processes are distinguished because they have a key role in "delay insensitive" implementations. Informally, a process is endochronous on a set of variables I if, given an external (asynchronous) stimulation of I, it is capable of reconstructing a unique behavior (up to stretch-equivalence). The following definition makes this more formal.

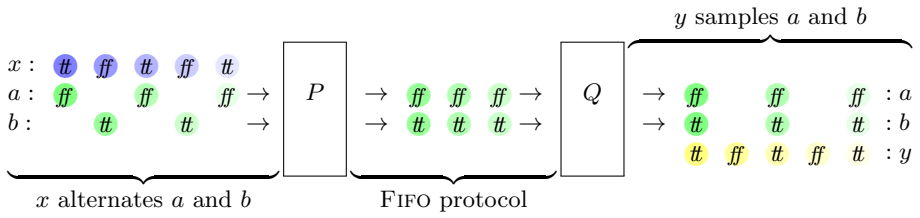


Figure 4: Asynchrony as the relaxation of synchronization relations. Relaxation allows to individually stretch the signals a and b of flow-invariant processes P and Q. While P alternates a and b depending on the values of x, process Q instead needs to sample a or b depending on the values of y. Relaxation is implemented by the insertion of a FIFO protocol between P and Q. The FIFO itself can be defined as a flow-invariant process.

Definition 11 (Endochronous process). A process P defined on V is *endochronous* on $I \subseteq V$, $I = \{i_1, \ldots, i_n\}$, if and only if the function $\Phi : P \to P_{|\{i_1\}} \times \ldots \times P_{|\{i_n\}}$ such that $\Phi(b) = (b_{|\{i_1\}}, \ldots, b_{|\{i_n\}})$ is injective (and thus bijective, since it is necessarily surjective): $\forall b_1, b_2 \in P, \Phi(b_1) = \Phi(b_2) \Rightarrow b_1 = b_2$.

To examplify endochrony, consider an oversampling *write* process (that will be depicted at the end of Section 3), which is paced by a Boolean input signal c, such that $c_{\approx} = (ff, tt, ff, tt)$, to read an integer input y, such that $y_{\approx} = 1, 2, 3, 4$ when c is true and output either of that value or the previously memorized one (initially 0) along an output signal x. Whichever time tags t are assigned to c, x, y, y must always synchronizes to c and x to carry a new value of y if c is true, hence $x_{\approx} = (0, 2, 2, 4)$.

As a consequence of this definition, a process P defined on V is endochronous on $I \subseteq V$ iff $\forall b_1, b_2 \in P$, $b_{1|I} \approx b_{2|I} \Leftrightarrow b_1 \geqslant b_2$. If the set of variables I is the set of inputs of P, P is endochronous if it is endochronous on I. In other words, a process is endochronous if any behavior of this process is entirely determined by the sequence of values carried by I independently of their relative presence and absence.

Proposition 1. *A process which is endochronous on I is flow-invariant on I.*

Proof. Let P be an endochronous process on I, and b, c such that $(b_{|I}) \approx (c_{|I})$. From endochrony we have $b \geqslant c$, and from properties of flow-equivalence (cf. 2.3), $b \approx c$. □

While Proposition 1 is easy to prove for a trace semantics, thanks to the algebraic properties provided by the stretching and relaxation [**?**], it also has an inductive proof in a constructive operational semantics of Signal that materializes the role played by event and/or time triggers [**?**] upon which the evaluation of input flows by a given endochronous starts and converges to a unique solution. Endochrony will further be illustrated in Section 3.8.3 using an implementation of the example described in Section 2.10.

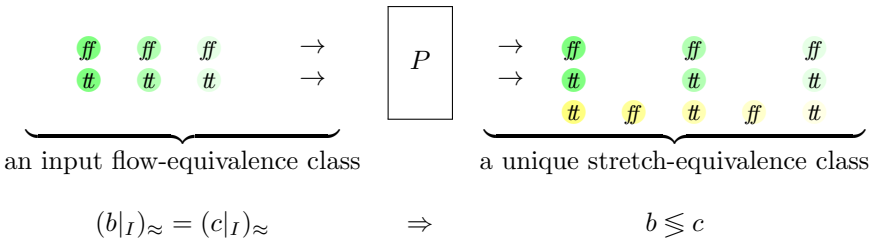


Figure 5: An endochronous process P processes two input flows (green) by building a unique stretch-equivalence class of behaviors that synchronize the inputs to the occurences of the value true on its own, private, clock (yellow).

2.6. Deterministic processes

We will say that a process is deterministic on a set of variables if any behavior of this process is entirely determined by its restriction to this set of variables. More formally, we set this definition:

Definition 12 (Deterministic process). A process P defined on V is *deterministic* on $I \subseteq V$ if and only if the function $\Psi : P \to P_{|I}$ such that $\Psi(b) = b_{|I}$ is injective (and thus bijective, since it is necessarily surjective): $\forall b_1, b_2 \in P, \Psi(b_1) = \Psi(b_2) \Rightarrow b_1 = b_2$.

Hence, a process P defined on V is deterministic on $I \subseteq V$ iff $\forall b_1, b_2 \in P,\ b_{1|I} \lesseqgtr b_{2|I} \Leftrightarrow b_1 \lesseqgtr b_2$. A process is deterministic on a set of variables iff any two behaviors of this process are stretch-equivalent when their projections on this set of variables are stretch-equivalent. We will say that a process is deterministic if it is deterministic on I.

To examplify determinism, now consider a downsampling *read* process, which is paced by a Boolean input signal c, such that $c_{\approx} = (ff, tt, ff, tt)$, and synchronizes to an integer input signal x, such that $x_{\approx} = 1, 2, 3, 4$. When c is true, the value of x is output along a signal y. Hence, the output $y_{\approx} = (2, 4)$, at the exact same time tags than the even occurrences of c and x.

The following proposition is easily deduced from the definitions of endochronous processes and deterministic processes:

Proposition 2. *If a process P defined on V is endochronous on $I \subseteq V$, then it is deterministic on I.*

Examples of deterministic and non-deterministic Signal processes will be presented in Section 3.5.

2.7. Weakly endochronous processes

As endochrony is not preserved by composition, it may be desirable, to achieve compositionality, to define a weaker notion called *weak endochrony*. This requires a definition of the union of so-called *independent reactions*. Two

reactions are independent iff they have disjoint domains (they are defined on distinct variables).

The union $r \sqcup s$ of two independent non empty reactions r and s is defined as follows if they have the same tag, $tag(r) = tag(s)$:
$\forall x \in vars(r) \cup vars(s)$, $(r \sqcup s)(x) =$ if $x \in vars(r)$ then $r(x)$ else $s(x)$.

Definition 13 (Weakly endochronous process). A process P defined on V is weakly endochronous on $I \subseteq V$ iff:

1. P is deterministic on I;
2. for all behaviors b and independent reactions r and s and r and t, P satisfies:
   - if $b \cdot r \cdot s \in P$ then $b \cdot s \in P$,
   - if $b \cdot r \in P$ and $b \cdot s \in P$ then $b \cdot (r \sqcup s) \in P$,
   - if $b \cdot (r \sqcup s),\ b \cdot (r \sqcup t) \in P$ then $b \cdot r \cdot s,\ b \cdot r \cdot t \in P$.

P is weakly endochronous if it is weakly endochronous on its set of inputs.

Informally, a process is weakly endochronous iff it is deterministic and can perform independent reactions in any order. Note that endochrony implies weak endochrony.

A process P is *non-blocking* iff for all behaviors $b \in P$, there exists some reaction $r \in P$ such that $b \cdot r \in P$.

From its definition, it appears that checking weak-endochrony requires an exhaustive state-space exploration, which could be very costly, or would require a program analysis to abstract the state space to be explored, as in [22]. Determining that a process is weakly endochronous is even non decidable in general for infinite-state systems. A method has been proposed based on the static abstraction and analysis of clock and scheduling relations, allowing to model-check weak endochrony [25]. However, this can be still costly and it may be useful to define a property which is stronger than weak-endochrony and easy to check. This property will be called *polyendochrony*. Unlike endochrony, that allows to reconstruct, from a given set of signals, a unique behavior (modulo stretching) on the whole set of variables, polyendochrony allows to reconstruct a unique behavior (modulo stretching) on subsets of variables.

Subdomain of a variable in a process. Let P be a process on a set of variables V and r a (referent) variable in V. We call subdomain of r, denoted $\curlywedge r$, the set of variables for which associated signals are present only at some instants of r:
$\curlywedge r = \{x \in V | \forall b \in P, tags(b(x)) \subseteq tags(b(r))\}$.

For a behavior b, $b_{|\curlywedge r}$ is the projection of b on the subdomain of r.

Polyendochronous processes.

Definition 14 (Polyendochronous process). A process P defined on V is polyendochronous on $I \subseteq V$ iff
$\forall b_1, b_2 \in P$, $\forall r \in V$, $b_{1|I} \approx b_{2|I} \Leftrightarrow b_{1|\curlywedge r} \gtrless b_{2|\curlywedge r}$.

A process is polyendochronous if it is polyendochronous on I. Note that endochrony is a special case of polyendochrony: an endochronous process is a polyendochronous process for which all variables belong to the same subdomain.

Proposition 3. *A process which is polyendochronous on I is flow-invariant on I.*

PROOF. Let P be a polyendochronous process on I, and b, c such that $(b_{|I}) \approx (c_{|I})$. Polyendochrony ensures that for each $x \in vars(P)$ we have $b_{|\curlywedge x} \gtrless c_{|\curlywedge x}$. Thus $(b_{|\{x\}}) \gtrless (c_{|\{x\}})$. And from properties of flow-equivalence (cf. 2.3), $b \approx c$. □

Lemma 1 (Lemma of tag supports). *Let P be a polyendochronous process on I, for each behavior b of P, $tags(b) = tags(b_{|I})$.*

PROOF. Suppose there is a variable $r \in vars(P)$, a behavior $b \in P$ and a tag $\theta \in tags(b_{|\curlywedge r})$ such that $\theta \notin tags(b_{|I})$. The behavior prefix $b_{\leq\theta}$ is not stretch-equivalent to the behavior prefix $b_{<\Theta}$. However, prefixes $(b_{|I})_{\leq\theta}$ and $(b_{|I})_{<\theta}$ are stretch-equivalent since θ does not belong to $tags(b_{|I})$. Thus, for $\Theta = \{t \leq \theta\}$ and $\Theta' = \{t < \theta\}$, we have $(b_{|I})_{\leq\Theta} \approx (b_{|I})_{\leq\Theta'}$ and $(b_{|\curlywedge r})_{\leq\Theta} \not\gtrless (b_{|\curlywedge r})_{\leq\Theta'}$. This contradicts the hypothesis of polyendochrony, thus, $\forall t \in tags(b)$, $t \in tags(b_{|I})$.

In the other side, it is obvious that $\forall t \in tags(b_{|I})$, $t \in tags(b)$. □

Proposition 4. *A process which is polyendochronous on I is deterministic on I.*

PROOF. Consider behaviors b, c such that $b_{|I} \gtrless c_{|I}$: behaviors b and c are stretch-equivalent on I. Thus their tag supports restricted on I are isomorphic, and from lemma of tag supports, tag supports of the whole behaviors are also isomorphic. Moreover, we have $b_{|I} \approx c_{|I}$. Thus, from polyendochrony, we get $\forall r \in vars(r)$, $b_{|\curlywedge r} \gtrless c_{|\curlywedge r}$. In conclusion, tag supports of behaviors b and c are isomorphic and the signal values are preserved between b and c. Thus $b \gtrless c$. □

An example of a poly-endochronous Signal process is given in Section 3.9.

Relation to latency-insensitive protocols.

In [23], a theory of latency-insensitive protocols (LIP) is presented as a foundation of a methodology to design very large digital systems by assembling blocks of existing *intellectual property* (IP). In the theory of LIP, a process p may accept *stall moves* (a stall move consists of stretching a signal s of a given behavior b by one logical instant) and respond by the appropriate *procrastination* effect (the other signals s' of the behavior b will respond to the delay of s by making coordinated stall moves).

The mechanism of causally related stall moves is made explicit (and generalized to polychronous signals) in the relation of stretching. A stretch corresponds to a slide of tags in a given behavior: if an event e is stalled, then all other events e', causally related to e by the partial order relation $\leq$) will stall further. As a result, SIGNAL processes are *patient*, in the sense of [23].

Definition 15 (patience [23]). A patient process p is a process such that, for all behaviors $b \in p$, for all and for all signals $x \in (b)$ s.t. $b(x) = (t_j, v_j)_{j>0}$, if a signal $s' = (t'_k, v_k)_{k>0}$ differs from s by one stall move (i.e. there exists $l > 0$ s.t. $t_j = t'_k$ for all $j = k < l$ and $t_l \leq t'_l$) then there exists $b' \in p$ such that $b'(x) = s'$.

The (multi-clocked) notion of flow-equivalence relates to the notion of latency-equivalence of Carloni et al. [23]. Two signals are latency-equivalent iff they present the same values in the same order.

Definition 16 (latency-equivalence [23]). Let $(t_i)_{i>0}$ and $(t'_i)_{i>0}$ be two chains of s.t. for all $j, k > 0$, $j \leq k$ iff $t_j \leq t_k$ and $t'_j \leq t'_k$. The signals $s = (t_i, v_i)_{i>0}$ and $s' = (t'_i, v'_i)_{i>0}$ are latency-equivalent iff for all $i > 0$, $v_i = v'_i$.

Notice that, if s and s' are flow-equivalent, then they are latency equivalent. Flow-equivalence extends the property of latency-equivalence to multi-clocked systems: the theory of LIP considers synchronous processes equipped with totally ordered time tags (i.e. single-clocked systems). This hypothesis corresponds to the low-level behavior of circuits (where silence, i.e. τ events, corresponds to stall moves or "don't care" moves). The tagged model of polychronous signals considers a more general structure of partially ordered semi-lattice.

2.8. Signal operators

The Signal language provides a syntax to processes defined from particular relations between timed histories (signals) associated with *signal variables*. Elementary processes are defined by equations. An equation associates with a signal variable an expression built on operators over timed histories associated with signal variables. Syntactically, an equation is written $x := f(y_1, \ldots, y_n)$. It is said to define the output x and use the inputs $y_1, \ldots, y_n$ (minus x). The outputs of a process P are the variables it defines and its inputs are the other it uses. For a process denoted by P, we write $[\![P]\!]$ to denote the set of behaviors of the process.

Two kinds of operators are distinguished: synchronous operators and polychronous operators.

Synchronous operators.

Stepwise extension of function. Let f designate some n-ary total function $f : \mathbb{D}_1 \times \ldots \times \mathbb{D}_n \to \mathbb{D}_{n+1}$ where $\mathbb{D}_i \subseteq \mathbb{D}$ and $n > 0$ (such functions are, for instance, usual logic, arithmetic or relational operations). In Signal, these functions are extended to timed histories:

$$[\![x := f(y_1, \ldots, y_n)]\!] =$$

$$\left\{ b \in \mathcal{B}_{|\{x, y_1, \ldots, y_n\}} \;\middle|\; \begin{array}{l} tags(b(x)) = tags(b(y_1)) = \ldots = tags(b(y_n)) = C \in \mathcal{C} \\ \forall t \in C, b(x)(t) = f(b(y_1)(t), \ldots, b(y_n)(t)) \end{array} \right\}$$

Delay. The delay operator represents a state initialized with some value $v \in \mathbb{D}$:

$$[\![x := y\$ \ init \ v]\!] = \left\{ b \in \mathcal{B}_{|\{x,y\}} \left| \begin{array}{l} tags(b(x)) = tags(b(y)) = C \in \mathcal{C} \setminus \emptyset \\ b(x)(\min(C)) = v \\ \forall t \in C \setminus \min(C), b(x)(t) = b(y)(\mathrm{pred}_C(t)) \end{array} \right. \right\}$$

Polychronous operators. Polychronous operators relate timed histories defined on different sets of tags.

Sampling. The signal variable z, which takes its values in the Booleans $\mathbb{B}$, true tt and false ff serves as sampling condition:

$$[\![x := y \ when \ z]\!] =$$

$$\left\{ b \in \mathcal{B}_{|\{x,y,z\}} \left| \begin{array}{l} tags(b(x)) = \{t \in tags(b(y)) \cap tags(b(z)) | b(z)(t) = tt\} \\ \forall t \in tags(b(x)), b(x)(t) = b(y)(t) \end{array} \right. \right\}$$

Merge. The merge gives priority to its first operand:

$$[\![x := y \ default \ z]\!] =$$

$$\left\{ b \in \mathcal{B}_{|\{x,y,z\}} \left| \begin{array}{l} tags(b(x)) = tags(b(y)) \cup tags(b(z)) = C \in \mathcal{C} \\ \forall t \in C, b(x)(t) = \left\{ \begin{array}{l} b(y)(t) \text{ if } t \in tags(b(y)) \\ b(z)(t) \text{ if } t \notin tags(b(y)) \end{array} \right. \end{array} \right. \right\}$$

2.9. Signal syntax and derived operators

The concrete syntax for the composition of processes P and Q is $(|\ P \mid Q\ |)$.

For a process P defined on a set of variables V, the restriction of P on $V \setminus \{x\}$ (x is hidden from the outside of P) is denoted by P / x or P *where* x.

Derived operators are defined from the operators of the kernel language, defined in Section 2.8, with composition and restriction. The following derived operators will be used in the examples of this article (a complete definition of the language is available online [26]).

- Signal clock: $x := \hat{}y$ is defined by $x := y = y$ (y is present and tt when x is present: y has the type *event*, the single value of which is the value tt—this type is used to represent *clocks* of signals).
- Selection: $x :=$ *when* y (a.k.a. $x := [: y]$) is defined by $x := \hat{}y$ *when* y (x is a signal of type *event*, present—and tt—when the Boolean signal y is present and tt).
- Clock intersection: $x := y \ \hat{}* \ z$ is defined by $x := \hat{}y$ *when* $\hat{}z$ (x is a signal of type *event*, present when both signals y and z are present).
- Clock union: $x := y \ \hat{}+ \ z$ is defined by $x := \hat{}y$ *default* $\hat{}z$ (x is a signal of type *event*, present when either signal y or z is present).

- Clock difference: $x := y \hat{-} z$ is defined by $x := \mathit{when}\ ((\mathit{not}\ \hat{}z)\ \mathit{default}\ \hat{}y)$ (x is a signal of type *event*, present when the signal y is present but z is not).
- Null clock: $\hat{}0$ is defined by $\mathit{when}\ (\mathit{not}\ \hat{}x)$, x being any signal ($\hat{}0$ is a signal of type *event* which is never present).
- Synchronization: $x \hat{=} y$ is defined by $(c := \hat{}x = \hat{}y)/\ c$ (it is an output-free process which constrains signals x and y to be present at the same instants—signals which are present at the same instants are *synchronous*: they have the same clock).
- Clock relation: $x \hat{<} y$ is defined by $x \hat{=} x \hat{*} y$ (it is an output-free process which constrains the clock of the signal x to be less frequent—or equal to—that of the signal y).
- Partial definition: $x ::= y$ is defined by $x := y\ \mathit{default}\ x$ (x is defined by y when y is present, otherwise it may be present but its value is left undefined—multiple partial definitions of x may be provided).

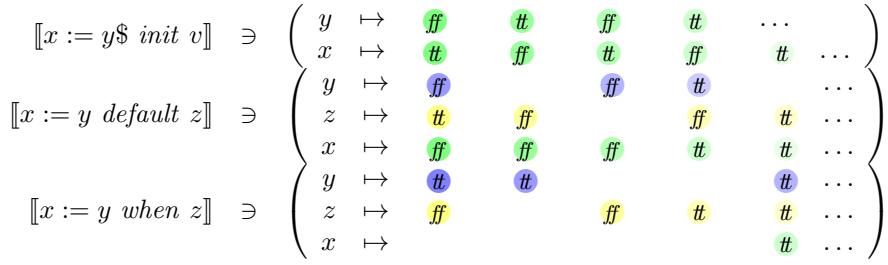


Figure 6: A depiction of some possible behaviors of delay, sampling and merge equations.

2.10. Examples of Signal programs

We use the Signal syntax with its modularity features allowing to declare a given process (a set of equations) as a *process model* with a name, possible static parameters (not used here) and explicit interface (input and output signals respectively preceded by "?" and "!").

Asynchronous memory cell. The following program describes an asynchronous memory cell with input x and output y:

```
process memoryCell = ( ? integer x;
                       ! integer y; )
  (| v := x default (v$ init 0)
   | y := v when ^y
   |)
```

```
   where integer v;
end;
```

The signal v representing the cell is hidden so that it is not possible to synchronize it from the outside: this makes the process asynchronous.

Integers generator. The following program, with no input, generates the sequence of positive integers:

```
process integersGenerator = ( ?
                              ! integer x; )
   (| x := (x$ init 0) + 1
    |);
```

Its clock is the clock of its output. It may be a node in a KPN.

Fibonacci sequence generator. In the same way, the following program describes a Fibonacci sequence generator:

```
process Fibonacci = ( ?
                      ! integer first; )
   (| first := second$ init 0
    | second := tmp$ init 1
    | tmp := first + second
    |)
   where integer second, tmp;
end;
```

It corresponds to the KPN represented as example in [27].

Internal master clock. The following program generates a sequence of n events after each occurrence of an input signal n, and before its next occurrence (n is supposed to be a positive or null integer signal):

```
process N_events = ( ? integer n;
                     ! event i; )
   (| cnt := n default (pre_cnt - 1)
    | pre_cnt := cnt$ init 0
    | n ^= when (pre_cnt = 0)
    | i := when (pre_cnt > 0)
    |)
   where integer cnt, pre_cnt;
end;
```

This program uses the "oversampling" mechanism: the "master clock" represents a refinement of time, with respect to the clock of the input signal. Here, this master clock is that of an internal signal (the common clock of *cnt* and *pre_cnt*).

3. Signal programs as KPNs

The formal definition of Kahn Process Networks is first recalled in Section 3.1 and some related works are mentioned in 3.2. In Section 3.3, we look at whether Signal operators can be considered as flow functions and in Section 3.4, we define an actor model for Signal operators. The question of deterministic versus non-deterministic Signal programs is considered in Section 3.5 and that of testing the absence in Section 3.6. Then, rules of syntactic equivalence allowing to rewrite Signal programs are provided in Section 3.7 and the formal clock calculus applied in compiling programs is described in Section 3.8. Resulting from these rewritings (in particular, the one of clock calculus), the first main result of this section is stated in Section 3.9, allowing to relate (composition of) endochronous processes and KPNs. In the rest of the section, special cases of such endochronous process networks are considered, including polyendochronous networks.

3.1. Fixed-point semantics of flow functions

Gilles Kahn's pioneering work in the seventies [6] has given semantical bases to the dataflow paradigm. Informally, Kahn Process Networks are process networks in which the nodes (which are concurrent sequential processes) communicate streams of data via FIFO channels with blocking read and non-blocking write operations. The denotational semantics of KPNs is defined as fixed-point semantics of flow functions. In this section, we mainly borrow to [27] the formulation of the semantics.

For a domain of values $\mathbb{D}$, we write $\mathbb{D}^\omega$ for the set of finite and countably infinite sequences of values in $\mathbb{D}$ (strings over alphabet $\mathbb{D}$). The empty sequence is denoted by ϵ. The prefix order on sequences is defined as follows: $\forall x \in \mathbb{D}^\omega, \epsilon \sqsubseteq x$, $\forall v \in \mathbb{D}, \forall x, y \in \mathbb{D}^\omega, x \sqsubseteq y$ iff $v \cdot x \sqsubseteq v \cdot y$ (where the symbol "$\cdot$" denotes the usual concatenation of strings). Intuitively, the order relates the amount of information. $(\mathbb{D}^\omega, \sqsubseteq)$ is a complete partial order (cpo). Each directed subset D in a cpo has a least upper bound (a subset is directed if it is non-empty and every pair of elements has an upper bound in the subset). Typically, an increasing chain of sequences $D = \{x_0, x_1, \ldots\}$, where $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, has a least upper bound. We write $\sqcup D$ for this supremum.

Let $(\mathbb{D}^\omega)^p$ be the set of p-tuples of sequences in $\mathbb{D}^\omega$. Tuples of sequences are ordered as well: $(X_1, \ldots, X_p) \sqsubseteq (X'_1, \ldots, X'_p)$ if $X_i \sqsubseteq X'_i$ for each i, $1 \leq i \leq p$. $((\mathbb{D}^\omega)^p, \sqsubseteq)$ is a cpo for any integer p.

A KPN process (or actor) with m inputs and n outputs is a function $f : (\mathbb{D}^\omega)^m \to (\mathbb{D}^\omega)^n$ that maps a tuple of input sequences into a tuple of output sequences. In a KPN, this function is required to be Scott-continuous, that is to say it preserves least upper bounds: for every directed subset D of the set of m-tuples of sequences in $\mathbb{D}^\omega$ (typically, an increasing chain of m-tuples of sequences), the image by f of the supremum of D is the supremum (that must exist) of the set of the images of all the elements of D. This is denoted by $f(\sqcup D) = \sqcup f(D)$. We say that f is a *flow function*. Intuitively, continuity expresses that any finite amount of information about the output of the function

is due to a finite amount of information about the input. Scott-continuity implies in particular monotonicity: $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$ (order preservation, or causality: the more input the function is given, the more output it produces, and computations flow from past to future). A process can be computed iteratively.

A KPN is defined by a set P of KPN processes and a set C of channels, partitioned into disjoint sets of inputs I, outputs O and internal channels L. Each process p has a set of input channels I_p and output channels O_p in C (with $\bigcup_{p\in P} I_p \cup \bigcup_{p\in P} O_p = C$ and $\bigcup_{p\in P} O_p = L \cup O$).

In the context of KPNs, a history of a channel associates with the channel (designated by a variable) the sequence of data (also referred to as stream, or sometimes *pure flow*) communicated along the channel. A history h of a set V of channels is a mapping from V to $\mathbb{D}^\omega$. The set of all histories of V is denoted as $H(V)$. Thus a KPN process p is described by a Scott-continuous function $f_p : H(I_p) \to H(O_p)$ that maps input histories to output histories.

For a history $h \in H(V)$, we write $h_{|V'}$ for the projection of h on a set of channels $V' \subseteq V$. We define an order on histories as follows: for two histories $h_1 \in V_1$, $h_2 \in V_2$ (with $V_1, V_2 \subseteq V$), $h_1 \sqsubseteq h_2$ iff $V_1 \subseteq V_2$ and for every $c \in V_1$, $h_1(c) \sqsubseteq h_2(c)$. The set of histories $(H(V), \sqsubseteq)$ is a cpo.

The behavior of a KPN is described by its network equations, expressing relations on histories $h \in C$ for all its processes: $h_{|O_p} = f_p(h_{|I_p})$ for all $p \in P$.

Then, it is described as a (Scott-continuous) flow function f from input histories to histories of all channels of the network, $f : H(I) \to H(C)$, that obeys to the following recursive equations:

$$\begin{cases} f(i)_{|I} = i \\ f(i)_{|O_p} = f_p(f(i)_{|I_p}) \text{ for all } p \in P \end{cases}$$

From these equations, we define a functional $\Phi : (H(I) \to H(C)) \to (H(I) \to H(C))$:

$$\begin{cases} \Phi(f)(i)_{|I} = i \\ \Phi(f)(i)_{|O_p} = f_p(f(i)_{|I_p}) \text{ for all } p \in P \end{cases}$$

Φ is a Scott-continuous function on a cpo. Thanks to Kleene fixed-point theorem, Φ has a least-fixed point: the "smallest" function f that satisfies the network equations. It is the supremum of the ascending Kleene chain of Φ: $\emptyset \sqsubseteq \Phi(\emptyset) \sqsubseteq \Phi(\Phi(\emptyset)) \sqsubseteq \ldots$ (where $\emptyset$ is the everywhere undefined function). The behavior of the network can be iteratively computed using Kleene iteration.

3.2. KPNs and related work

In Kahn's model, processes are deterministic. Considering non-deterministic computations is more problematic. It has been shown in [28] that, in this case, the input-output relation computed by a network is not compositional. A lot of references for non-deterministic operations may be found in [29]. In [30] the authors use trace sets, which are compositional, as abstractions of process behaviors and discuss the problem of having fair merge in dataflow networks.

On another side, synchronous dataflow languages Lustre [31] and Signal offer a way to abstract the time at which computations take place and allow to generate efficient implementations with bounded memory. Synchronous (or synchronized) dataflow appears as some restriction of general dataflow, which has

to respect some "synchronous constraints". The rules that define these restrictions provide a static analysis which is called "clock calculus" [15, 32]. Clock calculus is a verification of clock relations (timing relations between streams) in Lustre, while it is a synthesis of such relations in Signal (cf. Section 3.8).

In [33], the authors relate synchronous constraints to "listlessness" [34] and extend the class of static synchronous dataflow to higher order and some class of recursive networks. They call their programs "synchronous Kahn networks".

Another restricted dataflow model is that of SDF (for Synchronous Data Flow, again) [35], which is one of the basic models of computation of the Ptolemy framework [36, 37]. As it is the case for synchronous languages, SDF models have a "consistency checking" that makes possible static scheduling at compile time.

More generally, as discussed in [38], in order to give a formal semantics to heterogeneous systems, the authors of Ptolemy have proposed a semantics which is a generalization of a Kahn semantics based on a tagged signal model [39]. In Kahn's theory, the tag domain is $\mathbb{N}$, signals are partial functions from $\mathbb{N}$ to a set of values V, and signals are assumed to be prefix-closed (if they are undefined at some time n then they remain undefined for all $n' > n$). Generalization consists in considering other tag sets $\mathbb{T}$, where $\mathbb{T}$ is a poset, while remaining order-preserving [40]. In this theory, signals are partial functions from $\mathbb{T}$ to a set of values V. The authors of [38] observe that in many useful cases, simplifying assumptions can be considered: 1/ the tag set is a total order, 2/ the defined values of a signal can be indexed in non decreasing order (discrete signal). Then signals can be seen as streams and the theory reduces to the ordinary Kahn theory. Heterogeneity (dealing with different tag sets) and distribution are then handled in this generalized Kahn semantics.

On a practical aspect, KPNs are a popular model to describe signal processing and multimedia applications. Unlike synchronous dataflow models, general KPNs must be scheduled dynamically: a run-time system is necessary to schedule the processes and to manage memory for the channels. Conceptually, the FIFO channels of a KPN have an unbounded capacity, however, for a given realisation, a KPN has to run with a finite amount of memory. It is undecidable in general to know which buffer capacity is needed in every channel [41]. A model of execution has been proposed [42], where fixed capacities of memory are allocated to FIFOs and these bounds can be changed dynamically. This model is used for many implementations of KPNs [43, 44, 45]. In this model, a write action on a full FIFO is blocked until there is room again in the buffer. Unfortunately, if buffers are chosen too small, artificial deadlocks may appear and the proposed global deadlock resolution strategies do not guarantee fairness. In [46], it is proposed for a subclass of KPNs an improved scheduling strategy that uses local deadlock detection.

In the synchronous dataflow context, [47] proposes a model of "n-synchronous Kahn networks": two processes are n-synchronous if they can communicate in the ordinary synchronous model with a FIFO buffer of size n. Periodic clocks defined as periodic infinite binary words are considered. In this framework, minimal buffer sizes needed to communicate between synchronizable

clocks may be statically computed. The approach is generalized in [48] to non necessarily periodic clocks using a notion of "clock envelope" that represents interval of clocks up to bounded buffering.

3.3. Signal operators and flow functions

Signal operators can be viewed as actors that map a tuple of input sequences into an output sequence. Composed processes correspond clearly to networks of actors.

In particular, synchronous operators are flow functions.

Let $+_\omega$ be the stepwise extension on finite and countably infinite sequences of the usual + operator defined on some type domain. In Signal, constraints on length of flows (sequences) make this operator monotonic:

- $\forall s \in \mathbb{D}^\omega, s +_\omega \epsilon = \epsilon +_\omega s = \epsilon$
- $\forall v_1, v_2 \in \mathbb{D}, s_1, s_2 \in \mathbb{D}^\omega, v_1 \cdot s_1 +_\omega v_2 \cdot s_2 = (v_1 + v_2) \cdot (s_1 +_\omega s_2)$

Then, to prove Scott-continuity for a function f (representing here stepwise extension of usual functions or delay operator), we have to prove that for every chain $C = \{s_0, s_1, s_2, \ldots\}$ of partially constructed inputs of the actor f, $f(\sqcup_{s \in C}\ s) = \sqcup_{s \in C}\ f(s)$.

- $\sqcup_{s \in C}\ f(s) \sqsubseteq f(\sqcup_{s \in C}\ s)$:
  As $s \sqsubseteq \sqcup_{s \in C}\ s$, it follows directly from monotonicity since for each $s \in C$, $f(s) \sqsubseteq f(\sqcup_{s \in C}\ s)$.
- $f(\sqcup_{s \in C}\ s) \sqsubseteq \sqcup_{s \in C}\ f(s)$:
  Let $w \in f(C)$; w is some atomic piece of information in the output. It comes from some finite approximation, say u, of the input: $u \sqsubseteq \sqcup_{s \in C}\ s$. Thus $w \in f(u)$. Since u is finite and it is "included" in the supremum of the directed set, there is some v in C such that $u \sqsubseteq v$. By monotonicity, $f(u) \sqsubseteq f(v)$, thus $w \in f(v)$. And since $v \in C$, $f(v) \sqsubseteq \sqcup_{s \in C}\ f(s)$. □

On the other hand, Signal polychronous operators are not flow functions. If they are seen as operators on sequences of values, they are even not (mathematical) functions.

Consider for example (refering to the definition of the merge operator provided in Section 2.8) behaviors on a set of variables $\{x, y, z\}$, x being defined as $x := y$ *default* z, with different chains of tags for y:

- First, y has values 3, -2, -1, ... respectively associated with the chain of tags (in the time domain $\mathbb{N}$) $\{0, 1, 3, \ldots\}$, and z has values 0, 0, 2, 1, ... associated with the chain $\{1, 2, 3, 4, \ldots\}$. Then x has values 3, -2, 0, -1, 1, ..., associated with the chain of tags $\{0, 1, 2, 3, 4, \ldots\}$.
- Second, y has the same values, 3, -2, -1, ..., associated with the chain of tags $\{0, 1, 2, \ldots\}$, and z has also the same values as above, associated with the same chain. In this case, x has values 3, -2, -1, 2, 1, ..., associated with the chain of tags $\{0, 1, 2, 3, 4, \ldots\}$.

Considering only sequences of values, y and z have the same ones in both cases, but the sequences for x are different (non-determinism).

3.4. An actor model for deterministic reaction

With respect to the polychronous model presented in Section 2.1, the symbol $\perp$ is introduced to designate the absence of valuation of a signal: $\perp \notin \mathbb{D}, \mathbb{D}^{\perp} = \mathbb{D} \cup \{\perp\}$.

Definition 17 (Synchronized history, behavior). A *synchronized history* is a mapping s from a chain $C \in \mathcal{C}$ to the extended domain of values $\mathbb{D}^{\perp}$ ($s : C \to \mathbb{D}^{\perp}$).

A behavior on a finite set of variables V is a mapping b from V to the set of synchonized histories $\mathcal{S}$, such that all histories have the same domain (which is the time domain of b):
$b : V \to \mathcal{S}$, where $\forall x_1, \ldots, x_n \in V$, $tags(b(x_1)) = \ldots = tags(b(x_n)) = tags(b) = C \in \mathcal{C}$.

The symbol $\perp$ represents "meaningful" absence (Section 3.6).

Then, Signal operators (or actors) are expressed as flow functions and the semantics is defined as a fixed-point semantics.

Let $\mathbb{D}^{\omega}$ denote the set of sequences of values (extended streams, or *synchronized streams*, or sometimes *synchronized flows*) in $\mathbb{D}^{\perp}$ (the same notations are used for subsets of $\mathbb{D}$). Signal operators are defined as functions $f : \mathbb{D}_1^{\omega} \times \ldots \times \mathbb{D}_n^{\omega} \to \mathbb{D}_{n+1}^{\omega}$ where for all i, $\mathbb{D}_i \subseteq \mathbb{D}$.

These functions satisfy the following general rules:

- Termination: $(\exists i \in 1, \ldots, n, s_i = \epsilon) \Rightarrow f(s_1, \ldots, s_n) = \epsilon$
- Stretching: $f(\perp \cdot s_1, \ldots, \perp \cdot s_n) = \perp \cdot f(s_1, \ldots, s_n)$

A function f is *synchronous* iff it satisfies:
$\forall v_1, \ldots, v_n \in \mathbb{D}_1, \ldots, \mathbb{D}_n$, $x_1, \ldots, x_n \in \mathbb{D}_1^{\perp}, \ldots, \mathbb{D}_n^{\perp}$, $s_1, \ldots, s_n \in \mathbb{D}_1^{\omega}, \ldots, \mathbb{D}_n^{\omega}$,

$$\begin{cases} \bullet\ (\exists v \in \mathbb{D}_{n+1}, s \in \mathbb{D}_{n+1}^{\perp}) f(v_1 \cdot s_1, \ldots, v_n \cdot s_n) = v \cdot s \\ \text{and} \\ \bullet\ ((\exists i, j \in 1, \ldots, n)(\exists v \in \mathbb{D}_j)(x_i = \perp \wedge x_j = v)) \Rightarrow \\ \quad (f(x_1 \cdot s_1, \ldots, x_n \cdot s_n) = \epsilon) \end{cases}$$

Let a *configuration* of a set of variables be a simultaneous valuation (in $\mathbb{D}^{\perp}$) of these variables. The *absence configuration* of a set of variables is the configuration for which all variables have the absence value.

The so-called stretching rule corresponds to a possible insertion of any finite number of the absence configuration in a given behavior (stuttering). This defines an equivalence relation ("equivalence modulo $\perp$") on behaviors. Equivalence modulo $\perp$ preserves the simultaneousness of valuations within a configuration and the ordering of configurations within a behavior. This equivalence modulo $\perp$ corresponds to the stretch-equivalence of Section 2.2 and strict behaviors are those for which there is no absence configuration between two configurations which have valued variables (in $\mathbb{D}$).

3.4.1. Stream operators

Synchronous functions are defined as stream operators (the absence value $\perp$ serves only for stretching). They correspond to "delay insensitive" operations (the sequencing of computation is determined by the data flow).

Stepwise extension of function. Let f designate some n-ary total function: $f : \mathbb{D}_1 \times \ldots \times \mathbb{D}_n \to \mathbb{D}_{n+1}$. The corresponding actor, represented by its stepwise extension on streams, denoted by f_ω, is the synchronous function $f_\omega : \mathbb{D}_1^\omega \times \ldots \times \mathbb{D}_n^\omega \to \mathbb{D}_{n+1}^\omega$ that satisfies:
$\forall v_1, \ldots, v_n \in \mathbb{D}_1, \ldots, \mathbb{D}_n,\ s_1, \ldots, s_n \in \mathbb{D}_1^\omega, \ldots, \mathbb{D}_n^\omega,$
$\quad f_\omega(v_1 \cdot s_1, \ldots, v_n \cdot s_n) = f(v_1, \ldots, v_n) \cdot f_\omega(s_1, \ldots, s_n)$

Delay. The actor representing the delay is the synchronous function $delay : \mathbb{D}_1^\omega \times \mathbb{D}_1 \to \mathbb{D}_1^\omega$ that satisfies:
$\forall v_1, v_2 \in \mathbb{D}_1,\ s \in \mathbb{D}_1^\omega,$
$\quad delay(v_1 \cdot s, v_2) = v_2 \cdot delay(s, v_1)$

3.4.2. Extended stream operators

Polychronous operators are defined on synchronized streams, taking into account the absence value $\perp$.

Sampling. The actor representing the sampling operator is the function $when : \mathbb{D}_1^\omega \times \mathbb{B}^\omega \to \mathbb{D}_1^\omega$ that satisfies:
$\forall x_1 \in \mathbb{D}_1^\perp,\ s_1, s_2 \in \mathbb{D}_1^\omega,$

- $when(x_1 \cdot s_1, tt \cdot s_2) = x_1 \cdot when(s_1, s_2)$
- $when(x_1 \cdot s_1, ff \cdot s_2) = \perp \cdot when(s_1, s_2)$
- $when(x_1 \cdot s_1, \perp \cdot s_2) = \perp \cdot when(s_1, s_2)$

Merge. The actor representing the merge operator is the function $default : \mathbb{D}_1^\omega \times \mathbb{D}_1^\omega \to \mathbb{D}_1^\omega$ that satisfies:
$\forall v_1 \in \mathbb{D}_1, x_2 \in \mathbb{D}_1^\perp,\ s_1, s_2 \in \mathbb{D}_1^\omega,$

- $default(v_1 \cdot s_1, x_2 \cdot s_2) = v_1 \cdot default(s_1, s_2)$
- $default(\perp \cdot s_1, x_2 \cdot s_2) = x_2 \cdot default(s_1, s_2)$

Example. For the two behaviors of the merge operator considered in Section 3.3:

- y having successive values (including the absence value) 3, -2, $\perp$, -1, $\perp$, ... and z successive values $\perp$, 0, 0, 2, 1, ..., then x, defined as $x := y\ default\ z$, has successive values 3, -2, 0, -1, 1, ...

- y having successive values 3, -2, -1, $\perp$, $\perp$, ... and z successive values $\perp$, 0, 0, 2, 1, ..., then x has successive values 3, -2, -1, 2, 1, ...

It appears clearly that sampling and merge operators are now flow functions on the extended domain: they are *synchronized flow functions*. However, Signal programs may be non-deterministic. Thus they are not KPNs in general.

3.5. Deterministic and non-deterministic Signal programs

Note that the definition of determinism given in Section 2.6 takes into account the synchronizations between the considered variables (or, equivalently, the possible absence value for these variables).

As a consequence, all basic Signal equations (presented in 2.8) are deterministic. However, the determinism is not stable by composition and restriction.

Processes with feedback (typically, $x := f(x)$) or local variables (due to restriction) may be non-deterministic.

The input-free process $(|\ x := x\$\ init\ 0 \mid y := y\$\ init\ 1\ |)$ is a typical example of non-deterministic process. Both following output synchronized streams (represented here as sequences of pairs of values corresponding respectively to the outputs x and y) are examples of possible behaviors: $(0,1),(\perp,1),(0,\perp),(\perp,1),\ \ldots$ and $(\perp,1),(0,\perp),(0,\perp),(\perp,1),\ \ldots$ (since x holds a sequence of constant values 0 separated by any number of $\perp$ and y holds a sequence of constant values 1 also separated by any number of $\perp$). However, if we consider only streams (without the absence value), this process describes a flow function.

On the contrary, the process $(|\ x := a \mid y := b\ |)$ (where a and b are two independent inputs) is deterministic (but it is not endochronous).

The process with trivial cycle $x := a\ or\ x$ is non-deterministic since x can be *tt* or *ff* when a is *ff*.

The following process is also non-deterministic: $(|\ x := a\ default\ ((x\$\ init\ 0) + 1) \mid b := x\ when\ \hat{}b\ |)/x$. The internal signal x holds the value of the input a when there is an occurrence of a and then successive increments of this value (until the next occurrence of a). But since x is hidden from the outside, it is not possible to synchronize it, thus it can have any number of occurrences between successive occurrences of a, and b can output any of its values.

Other examples of non-deterministic processes are provided by partial definitions such as $x ::= a$ (equivalent to $x := a\ default\ x$): it only defines x asï¿½being equal to a when a is present; x may be present at other instants but its values at these instants are left undefined in this process. A given signal may have multiple definitions, obtained using several partial definitions $(|\ x ::= a_1 \mid \ldots \mid x ::= a_n\ |)$ (note that in KPNs, no two processes send data to the same channel). Such multiple definitions are consistent if for every pair of equations $x ::= a_i$, $x ::= a_j$, when a_i and a_j are both present, they hold the same value. Technically, if one can prove that $when\ ((a_1\ when\ \hat{}a_2) = (a_2\ when\ \hat{}a_1))\ \hat{} = \ a_1\ \hat{}*\ a_2$, then the process $(|\ x ::= a_1 \mid x ::= a_2\ |)$ is equivalent to $x ::= a_1\ default\ a_2$.

3.6. Meaningful absence

Kahn Process Networks are characterized by their determinacy. In the execution model of a KPN, writes to the channels are nonblocking, but reads are blocking. When a read operation is executed on an empty input channel, the reading process should stall until the FIFO has sufficient tokens. It cannot do anything else, for example testing for absence, otherwise it would break determinacy.

Although Signal operators can be seen as flow functions on the extended domain, the additional value is an encoding for the absence, thus it may be necessary to test this absence value to get the result. This is the case, for example, for the merge operator $x := y$ *default* z. If we add to this process two synchronous Boolean inputs, cy and cz, representing respectively the clocks of y and z (cy, resp. cz, is tt iff y, resp. z, is present), we get the following process (written in Signal syntax):

```
process merge = ( ? boolean cy, cz; integer y, z;
                  ! integer x; )
  (| x := y default z
   | cy ^= cz
   | y ^= when cy
   | z ^= when cz
   |);
```

The process *merge* is a flow function. It can be implemented without testing the absence (test for absence is replaced by a test to the value *ff* of the added Boolean inputs).

More generally, adding Boolean signals as inputs may be a way to transform a process into a *synchronized* flow function that does not need to test the absence of signals.

3.7. Syntactic equivalence

Syntactic equivalence rules apply to Signal programs. In particular, the following properties of composition and restriction are used to rewrite programs.

- Associativity: $(|\ P \mid Q\ |) \mid R \equiv P \mid (|\ Q \mid R\ |)$.
- Commutativity: $P \mid Q \equiv Q \mid P$.
- Idempotence: if P is a process without "side effects", then $P \mid P \equiv P$ (in the full Signal language, side effects may be produced by calls to external system functions, for instance).
- Externalization of restrictions: if x is not a signal of P, then $P \mid Q \,/\, x \equiv (|\ P \mid Q\ |) \,/\, x$.

As a consequence, expressions of composed processes can be normalized, modulo required substitutions of variables, as the composition of elementary processes, included in terminal restriction.

Expressions on signals can also be normalized thanks to properties of the operators.

- *when* is associative and right commutative:
  $(x \ when \ y) \ when \ z \ \equiv \ x \ when \ (y \ when \ z) \ \equiv \ x \ when \ (z \ when \ y)$.
- *default* is associative:
  $(x \ default \ y) \ default \ z \ \equiv \ x \ default \ (y \ default \ z)$.
- *default* can be written in exclusive normal form:
  $x \ default \ y \ \equiv \ x \ default \ (y \ when \ (y \ \hat{-} \ x))$.
- *default* commutes in exclusive normal form:
  if $x \ \hat{*} \ y \ \hat{=} \ \hat{}0$ (x and y are exclusive), then $x \ default \ y \ \equiv \ y \ default \ x$.
- *when* is right distributive over *default*:
  $(x \ default \ y) \ when \ z \ \equiv \ (x \ when \ z) \ default \ (y \ when \ z)$.

Then every signal x can be rewritten as follows:
$x := (e_1 \ when \ c_1) \ default \ \ldots \ default \ (e_n \ when \ c_n) \ default \ ((x\$ \ init \ v) \ when \ c_{n+1})$,
where the e_i are synchronous expressions with no delay and the c_i are signals of type *event* representing mutually exclusive clocks (c_{n+1} can be the null clcok).

As a fundamental characteristic, the equational nature of the Signal language makes it possible to consider the compilation of a process as a composition of endomorphisms over Signal processes.

3.8. Clock calculus

The *clock calculus* [32] is one of the formal tools allowing to rewrite Signal processes, in order to make explicit synchronization relations and to organize the "control" of processes as a *clock hierarchy*.

3.8.1. Clock relations

Synchronisation relations (or clock relations) associated with basic Signal expressions P are described in Table 1 as Signal expressions $\mathcal{C}l(P)$. The notations $[: z]$ and $[/: z]$ are other forms for, respectively, *when* z and *when not* z (which are signals of type *event* representing clocks).

Moreover, for every Boolean signal b, the following clock relations are stated:

$$\begin{array}{lll} [: b] \ \hat{+} \ [/: b] & \hat{=} & b \\ [: b] \ \hat{*} \ [/: b] & \hat{=} & \hat{}0 \end{array}$$

(the clock of a Boolean signal b is partitioned into its exclusive subclocks $[: b]$ and $[/: b]$, which represent the instants at which the signal b is present and has, respectively, the values *tt* and *ff*).

The so defined transformation $\mathcal{C}l$ satifies the following property making $\mathcal{C}l(P)$ an abstraction of P:

$$(| \ \mathcal{C}l(P) \ | \ P \ |) \ \equiv \ P.$$

Table 1: Clock relations associated with a process

Process P	Clock relations $\mathcal{C}l(P)$
$x := f(y_1, \ldots, y_n)$	$x \hat{=} y_1 \hat{=} \ldots \hat{=} y_n$
$x := y\$ \ init \ v$	$x \hat{=} y$
$x := y \ when \ z$	$x \hat{=} y \hat{*} [: z]$
$x := y \ when \ not \ z$	$x \hat{=} y \hat{*} [/: z]$
$x := y \ default \ z$	$x \hat{=} y \hat{+} z$
$(\mid P \mid Q \mid)$	$(\mid \mathcal{C}l(P) \mid \mathcal{C}l(Q) \mid)$
$P \ / \ x$	$\mathcal{C}l(P) \ / \ x$

Note that the variables involved in clock relations are "clock variables" $\hat{}x$: for instance, $x \hat{=} y \hat{+} z$ can be written equivalently $\hat{}x \hat{=} \hat{}y \hat{+} \hat{}z$. Clock variables can be seen as Boolean variables (as a Boolean variable, $\hat{}x$ will be denoted by $\hat{x}$) related in a system of Boolean equations. Clock operators $\hat{*}$, $\hat{+}$, $\hat{-}$ correspond respectively to Boolean operations of conjunction ($\wedge$), disjunction ($\vee$) and difference ($x \setminus y = x \wedge \neg y$, where $\neg$ is the negation).

3.8.2. Clock hierarchy

From clock relations, the clock calculus has to determine how clocks can be computed. For that purpose, it builds a (well-formed) *clock hierarchy* and tries to define clocks with Boolean functions.

Definition 18 (Clock hierarchy). A clock hierarchy is a relation denoted by $\hat{\searrow}$ (*dominate*) on the set of clocks (a given clock being associated with synchronous signals). A clock c dominates a clock d ($c \hat{\searrow} d$) if d is computed as a function of Boolean signals the clocks of which are c and/or clocks recursively dominated by c (d is hierarchically lower than c).

The definition of a well-formed clock hierarchy is made more precise below.

Initially, each clock is an isolated node of the clock hierarchy (it is a root of an elementary clock tree reduced to this node).

To reduce the number of trees by placing clocks in the clock hierarchy, the clock calculus considers a set F of Boolean signals the value of which is considered "free" (for the clock calculus): input Boolean signals, delayed Boolean signals, Boolean signals defined by non-Boolean predicates (for example, $c := a > b$). Let $c \in F$ be such a free variable. The clock of c, $\hat{c}$, is partitioned into its subclocks $[: c]$ and $[/: c]$. This partition is represented by a tree by adding the following elements in the *dominate* relation (which is an inclusion relation): $\hat{c} \hat{\searrow} [: c]$ and $\hat{c} \hat{\searrow} [/: c]$ (note that if cï¿½is defined by $a > b$ for instance, then $\hat{c} = \hat{a} = \hat{b}$). Generally, let $d \in F$ be a free variable that partitions a clock $\hat{c}_1$ ($\hat{c}_1 = \hat{d}$), the following relations apply: $\hat{c}_1 \hat{\searrow} [: d]$ and $\hat{c}_1 \hat{\searrow} [/: d]$. The depth of the hierarchy may grow since the clock $\hat{c}_1$ may be itself dominated by another clock.

Then, clocks written as Boolean formulas (using operators $\wedge$, $\vee$ and $\backslash$) are set in the clock hierarchy according to the following rules ($(\hat{\searrow})^*$ designates the transitive closure of $\hat{\searrow}$):

- If $\hat{x} = f(\hat{x}_1, \hat{x}_2)$, where f is a Boolean operator, and there exists c such that $c\ (\hat{\searrow})^*\ \hat{x}_1$ and $c\ (\hat{\searrow})^*\ \hat{x}_2$ ($\hat{x}_1$ and $\hat{x}_2$ are placed in the same tree of the clock hierarchy), then $\hat{x}$ is written in canonical form $\hat{x} = g(\hat{y}_1, \ldots, \hat{y}_n)$ (using BDDs) [49]. The clock $g(\hat{y}_1, \ldots, \hat{y}_n)$ is either the null clock, or the clock c, or is transitively dominated by c. If such a c does not exist, $\hat{x}$ remains as a root of the clock hierarchy (with an associated clock formula—it is not a free clock).

- If $\hat{x} = g(\hat{y}_1, \ldots, \hat{y}_n)$ is a canonical form with $n \geq 2$ and there exists d such that $d\ (\hat{\searrow})^*\ \hat{y}_i$ for all $\hat{y}_i$ in $\hat{y}_1, \ldots, \hat{y}_n$, then there exists a lowest clock c that dominates those $\hat{y}_i$, and the following relation is added: $c\ \hat{\searrow}\ \hat{x}$. The direct parent of the clock $\hat{x}$ in the tree is the least common ancestor of clocks $\hat{y}_i$ (the insertion of a formula into a tree can be seen as a factorization).

This again reduces the number of trees.

A clock hierarchy constructed with these rules is well-formed in the following sense:

Definition 19 (Well-formed clock hierarchy). A clock hierarchy is ill-formed iff either there exists c such that $[:\ c]\ \hat{\searrow}\ \hat{c}$ or $[/:\ c]\ \hat{\searrow}\ \hat{c}$, or there exists $b_1 = f(c_1, c_2)$, where f is a Boolean operator, and b_2, such that $b_1\ \hat{\searrow}\ b_2$, where $b_2\ \hat{\searrow}\ c_1$, $b_2\ \hat{\searrow}\ c_2$ and $\forall b$ such that $b\ \hat{\searrow}\ c_1$ and $b\ \hat{\searrow}\ c_2$, $b\ \hat{\searrow}\ b_2$. A well-formed clock hierarchy is a clock hierarchy which is not ill-formed.

Let $\min \hat{\searrow}$ denote the set of minimal elements for the relation $\hat{\searrow}$. The roots of the clock hierarchy (clocks which have no higher clock) are the clock equivalence classes of minimal elements for the relation $\hat{\searrow}$:
${}^0\hat{\searrow} = \{c_\sim \mid c \in \min \hat{\searrow}\}$.

A tree of root c in the clock hierarchy, denoted by ${}^c\hat{\searrow}$, is the tree from c in the relation $\hat{\searrow}$:
${}^c\hat{\searrow} = \{(c, d)\} \cup {}^d\hat{\searrow}$ such that $c\ \hat{\searrow}\ d$.

Definition 20 (Hierarchical process). A process is *hierarchical* iff its clock hierarchy has a unique root.

Canonical forms are given to clock formulas placed in a given tree. The roots which have no associated clock formula are considered free. Concerning roots with associated formula, classical Boolean reductions are applied on these formulas in order to merge trees and to iterate, if possible, the "placing" process in the clock hierarchy. The rewriting of formulas is limited using a fixed maximal depth in the syntax of formulas to avoid termination problems.

In some contexts, the clock difference operator, which may appear in clock expressions, may be problematic since it refers to clock negation. However, even

if the canonical forms of clocks (in a given clock tree) use the clock difference operator, this operator could be easily eliminated and such a canonical form can be expressed as a sum of products of conditions placed in the considered tree (disjunctive form) [50]. On the other hand, if a clock formula is associated with a root and this clock formula is a clock difference, it is not possible to eliminate it without referring to the absence (the negation) of a clock. We say that the clock hierarchy is *disjunctive* if it has no root clock the expression of which is defined using clock difference.

Definition 21 (Well-clocked process). A process is *well-clocked* if its clock hierarchy is well-formed and disjunctive.

Note that the clock calculus does not reach completeness: it is rather a heuristic calculus aimed at efficient compilation (of a control flow graph). Thus, in some particular cases, it may fail to provide an explicit form for some clock equations.

When the clock hierarchy computed by the clock calculus is composed of a single tree (i.e., there is a unique clock k with no higher clock and such that all other clocks are lower than k), then it means that all clocks can be computed by explicit functions from the root clock. Thus the process has a fastest rated clock (which is the root clock) and the status (presence/absence) of all signals is expressed as a flow function. From their Boolean expression, clocks can be expressed as Booleans the presence of which is also expressed, recursively, by a Boolean; and the test for presence/absence is replaced by a test of value of a Boolean, which, thanks to the clock hierarchy (which represents an inclusion relation), is conditioned itself by the Boolean clock of this Boolean.

The equations defining the value of the signals are themselves conditioned by the Boolean clock of these signals. Thus, when the clock hierarchy is composed of a single tree, and provided that there is no cyclic definition, thanks to syntactic equivalence rules and clock calculus, the whole process is rewritten as a synchronized flow function and the absence value has never to be tested. Detection of cyclic definitions corresponding to possible "deadlocks" is briefly described below, Section 3.8.4 (refer to this section for the definition of an *acyclic process*—the intuition about that may be sufficient in the following).

As a consequence, the input streams of the process (the sequences of input values, independently of their relative presence/absence) are sufficient to completely determine the behavior, in synchronized histories, of the process.

Definition 22 (Compilable process). A process is *compilable* iff it is well-clocked and acyclic.

Proposition 5. *Let P be a compilable process. There exists $I \subset vars(P)$ such that P is deterministic on I.*

Proof. For $I = vars(P)$, the process P is immediately deterministic. We can remove from I the variables linked in all behaviors, P remains deterministic (x is linked if there exist $y_i..y_n \in I$ such that $x = f(y_i..y_n)$). □

Proposition 6. *Let P be a process which is compilable and hierarchical. There exists $I \subset vars(P)$ such that P is endochronous on I.*

PROOF. Let $r_\sim$ be the single root of the clock hierarchy of P. P is compilable, thus it is deterministic on some set I. If a representative of the root, r, belongs to I, the result is immediate: r is present in each reaction, thus for any behavior b in P, $tags(b) = tags(b_{|\{r\}}) = tags(b_{|I})$. If no representative of the root, r, is is I, then r is linked: there exist $y_i..y_n \in I$ such that $r = f(y_i..y_n)$. Then $tags(b) = tags(b_{|\{y_i..y_n\}}) = tags(b_{|I})$. □

The roots of the clock hierarchy are also called *master clocks*. When its clock hierarchy has several master clocks, and there is no additional "clock constraint" (several formulas associated to a given clock, but which have not been proved equivalent), a process may be further parameterized so that it becomes endochronous: an example has been given in Section 3.6. This transformation can be named "endochronization", or "lustrage" (the resulting process can be seen as a Lustre program).

3.8.3. A small example

To illustrate clock calculus and its results, we consider as simple example the process *N_events* presented in Section 2.10.

First, the clock calculus establishes clock relations and applies elementary clock reductions. The process is rewritten as the following set of equations (texts between "%" are comments, *IOStepN_events* is the union of the clocks of the interface signals):

```
(| (| CLK_8 ^= cnt ^- [:pre_cnt=0] |)
 | (| % CLOCK of SUBTREE is represented by ^cnt %
      cnt ^= cnt ^+ [:pre_cnt=0]
    | cnt ^= pre_cnt
    | % SUBTREE of ^cnt %
      (| % ACTIONS at CLOCK ^cnt %
         (| cnt := n default ((pre_cnt-1) when CLK_8)
          | pre_cnt := cnt$ init 0
          |) |)
    |)
 | (| [:pre_cnt=0] ^= n |)
 | (| i ^= [:pre_cnt>0] |)
 | (| IOStepN_events ^= [:pre_cnt=0] ^+ i |)
 | (| CLK_17 ^= cnt ^+ IOStepN_events |)
 | (| CLK_18 ^= cnt ^+ CLK_17 |)
 | (| CLK_19 ^= [:pre_cnt=0] ^+ CLK_18 |)
 | (| CLK_20 ^= i ^+ CLK_19 |)
 |)
```

Then clock static resolution is applied and clock hierarchy is refined. The process is rewritten as follows:

```
% ENDOCHRONOUS: ROOT CLOCK is represented by ^cnt
  ENDOCHRONOUS with internal ROOT CLOCK
  CLOCK PARAMETERS: ^cnt %
(| (| cnt ^= pre_cnt
    | % SUBTREE of ^cnt %
      (| (| [:pre_cnt=0] ^= n |)
       | (| i ^= [:pre_cnt>0] |)
       | (| IOStepN_events ^= [:pre_cnt=0] ^+ i |)
       | % ACTIONS at CLOCK ^cnt %
         (| cnt := n default ((pre_cnt-1) when (not (pre_cnt=0)))
          | pre_cnt := cnt$ init 0
         |) |) |) |)
```

The clock hierarchy has a single root (which is here the clock of an internal signal): this clock $\hat{}cnt$ dominates clocks of n, i and *IOStepN_events*.

Finally, event clocks are transformed into Boolean clocks. The process is rewritten as follows (where, for example, the Boolean clock *C_IOStepN_events* is now written as a Boolean expression):

```
% ENDOCHRONOUS: ROOT CLOCK is represented by [:Tick]
  ENDOCHRONOUS with internal ROOT CLOCK %
(| (| [:Tick] ^= Tick
    | % SUBTREE of [:Tick] %
      (| [:Tick] ^= cnt ^= pre_cnt ^= L31_n ^= CLK_12
                 ^= L36_i ^= CLK_15 ^= C_IOStepN_events
       | (| [:L31_n] ^= n |)
       | (| [:L36_i] ^= i
          | % SUBTREE of [:L36_i] %
            (| % ACTIONS at CLOCK [:L36_i] %
               (| i := true when L36_i |) |)
         |)
       | % ACTIONS at CLOCK [:Tick] %
         (| cnt := n default ((pre_cnt-1) when CLK_12)
          | pre_cnt := cnt$ init 0
          | Tick := true when Tick
          | L31_n := pre_cnt=0
          | CLK_12 := not L31_n
          | L36_i := pre_cnt>0
          | CLK_15 := not L36_i
          | C_IOStepN_events := L31_n or L36_i
         |) |) |) |)
```

3.8.4. Acyclic process

Detection of cyclic definitions requires the building of a directed graph representing precedence relations between signals and clocks. Associated with the clock hierarchy, this "precedence graph" is the basic formal tool for the internal representation of programs. Precedence relations are conditioned by the clock at which the precedence holds: *x precedes y at clock $\hat{c}$* is represented as $\hat{c} : x \to y$. It expresses that for all instants of the clock $\hat{c}$, the computation of y cannot be performed before the value of x is known. Precedence relations associated with basic Signal expressions P are described in Table 2.

Process P	Precedence
$x := f(y_1, \ldots, y_n)$	$\hat{x} : y_1 \to x, \ldots, \hat{x} : y_n \to x$
$x := y\$ \ init \ v$	
$x := y \ when \ z$	$\hat{x} : y \to x$
$x := y \ default \ z$	$\hat{y} : y \to x, (\hat{z} \ \hat{-} \ \hat{y}) : z \to x$

Table 2: Precedence relations associated with a process

Moreover, for each signal x, there is a precedence $\hat{x} : \hat{x} \to x$, and for each clock $[: b]$ (respectively, $[/: b]$), there is a precedence $[: b] : b \to [: b]$ (respectively, $[/: b] : b \to [/: b]$). Note that no precedence is associated with the delay operator.

From basic precedence relations, a simple path algebra can be used to verify the absence of cycles:

$$\begin{cases} \hat{c} : x \to y \text{ and } \hat{d} : y \to z \Rightarrow (\hat{c} \ \hat{*} \ \hat{d}) : x \to z \\ \hat{c} : x \to y \text{ and } \hat{d} : x \to y \Rightarrow (\hat{c} \ \hat{+} \ \hat{d}) : x \to y \end{cases}$$

A precedence path $\hat{c}_1 : x_1 \to x_2$, $\hat{c}_2 : x_2 \to x_3$, $\ldots$, $\hat{c}_{n-1} : x_{n-1} \to x_1$ is called a *pseudo cycle*. Since precedences are conditioned by clocks, pseudo cycles are not necessarily *true cycles*: it may be the case that there is no common instant at which all the precedences along the cycle are effective. A process P is *acyclic*, or *deadlock-free*, iff for all pseudo cycles $\hat{c}_1 : x_1 \to x_2$, $\hat{c}_2 : x_2 \to x_3$, $\ldots$, $\hat{c}_{n-1} : x_{n-1} \to x_1$ in its associated precedence graph, the intersection of clocks $\hat{c}_1 \ \hat{*} \ \hat{c}_2 \ \hat{*} \ \ldots \ \hat{*} \ \hat{c}_{n-1}$ is equal to the null clock.

3.9. Process network

Networks of processes are constructed with (synchronous) composition of polychronous processes. The compositional design of networks requires for component processes and the way they communicate to verify composition properties.

Endochronous process network. The following result relates composition of endochronous processes and KPNs.

Composition of endochronous processes as KPN. .../... Notice the subtle "rewriting" condition: a Signal specification defines implicit clock and causality relations whose structure can both be used to verify the formal properties of its components (endochrony) but, once automatically inferred, can also be used to refine the original specification into one which, in particular, syntactically materializes its KPN structure (see Section 3.8.3).

Theorem 1. *A Signal specification denotes a Kahn Process Network if it can be rewritten, using syntactic equivalence and clock relations, as the composition of endochronous processes, whose master clocks are free or input signals.*

PROOF. This is a direct consequence of the results described in Section 3.8.2, which shows that an endochronous process is obtained from a Signal specification by an analytic rewriting that consists of a well-formed clock hierarchy (that defines its control flow) and an acyclic causality graph (that defines its static schedule). By Proposition 6, such a process is deterministic on a set of signals I which can be regarded as its input triggers. Now, plunge a set of such deterministic processes in an asynchronous network by interfacing their input triggers (the Is) with a idealized infinite FIFO buffers (which employ a blocking read method). By the definition of [6], they now form a deterministic process network over continuous flows of events. □

We observe, however, that the converse of Theorem 1 is not true: some Khan networks may not be rewritten as the asynchronous composition of endochronous Signal processes. This is essentially due to the limitations of the clock calculus, which uses symbolic graph and Boolean reasoning. Also, Theorem 1 does not express the fact that the fixpoint semantics of Signal processes always coincides with its "intersection" semantics, although it might, but only for deadlock-free processes.

Among the corollaries of Theorem 1, we further the discussion by considering classes of such endochronous process networks with particularly interesting formal properties that can be decided using Signal's clock calculus and, similarly, obtained by rewritting.

Endo-isochrony. Endo-isochrony is a property on the composition of endochronous processes. It ensures that a system composed of endochronous processes is equivalent in its synchronous and asynchronous composition.

Definition 23 (Endo-isochronous process). Let P_1 and P_2 be two processes defined respectively on V_1 and V_2. They are *endo-isochronous* iff P_1, P_2 and $(P_{1|V_1\cap V_2} \mid P_{2|V_1\cap V_2})$ are endochronous and $(P_{1|V_1\cap V_2} \mid P_{2|V_1\cap V_2})$ is acyclic.

Proposition 7. *If P_1 and P_2 are endo-isochronous, their asynchronous composition $P_1 \mid\mid P_2$ is flow-invariant.*

PROOF. If P_1 and P_2 are endo-isochronous then they are endochronous, thus $P_1 \mid\mid P_1$ and $P_2 \mid\mid P_2$ are flow-invariant. Since flow-invariance is compositional, $P_1 \mid\mid P_2$ is flow-invariant. □

Unfortunately, ensuring endo-isochrony may be costly on large systems of components since it requires to ensure endochrony for each component and for each pair of components. Hence the interest of considering polyendochronous processes (cf. Section 2.7).

Polyendochronous systems.

Proposition 8. *The synchronous composition of two polyendochronous processes P and Q is polyendochronous if the clock hierarchy of $P \mid Q$ is well-formed.*

PROOF. Let P and Q be two polyendochronous processes and b, c behaviors of $P \mid Q$. There exist b_P and $c_P \in P$ such that $b_{|vars(P)} = b_P$ and $c_{|vars(P)} = c_P$. Similarly, there exist b_Q and $c_Q \in Q$ such that $b_{|vars(Q)} = b_Q$ and $c_{|vars(Q)} = c_Q$. Since I is a subset of $vars(P) \cup vars(Q)$, we have $b_{P|I} \subseteq b_{|I}$, $b_{Q|I} \subseteq b_{|I}$, $c_{P|I} \subseteq c_{|I}$ and $c_{Q|I} \subseteq c_{|I}$.
The proposition $b_{|I} \approx c_{|I}$ implies $b_{P|I} \approx c_{P|I}$ and $b_{Q|I} \approx c_{Q|I}$.
P and Q are polyendochronous, thus, $\forall r \in vars(P)$, $\forall r' \in vars(Q)$, $(b_{P|\lambda r}) \gtrless (c_{P|\lambda r})$ and $(b_{Q|\lambda r'}) \gtrless (c_{Q|\lambda r'})$.
Or: $\forall r \in vars(P)$, $\forall r' \in vars(Q)$, $(b_{|vars(P)|\lambda r}) \gtrless (c_{|vars(P)|\lambda r})$ and $(b_{|vars(Q)|\lambda r'}) \gtrless (c_{|vars(Q)|\lambda r'})$.
For all $r \in vars(Q) \setminus vars(P)$, we have $(b_{|vars(P)|\lambda r}) \gtrless (c_{|vars(P)|\lambda r})$, since the sets $tags(b_{|vars(P)|\lambda r})$ and $tags(c_{|vars(P)|\lambda r})$ are empty.
Thus we can write: $\forall \rho \in vars(P \mid Q)$, $(b_{|vars(P)|\lambda \rho}) \gtrless (c_{|vars(P)|\lambda \rho})$ and $(b_{|vars(Q)|\lambda \rho}) \gtrless (c_{|vars(Q)|\lambda \rho})$.
Finally, if the clock hierarchy of $P \mid Q$ is well-formed, we can find all couples of variables of $P \mid Q$ such that $x \in \lambda y$.

- If x and y are both in $vars(P)$, then $tags(b(x)) \subseteq tags(b(y))$ for all behaviors $b \in P$.
- The same results are obtained for x, y in $vars(Q)$.
- If x is in $vars(P) \setminus vars(Q)$ and y in $vars(Q) \setminus vars(P)$, there exists a variable z in $vars(P) \cap vars(Q)$ such that $tags(b(x)) \subseteq tags(b(z))$ for all b in P and $tags(c(z)) \subseteq tags(c(y))$ for all c in Q. So for all d in $P \mid Q$ we have $tags(d(x)) \subseteq tags(d(y))$.
- And so on.

In all cases, it is shown that the variables of a subdomain use the tag support of their referent, namely:
$\forall \rho \in vars(P \mid Q)$, $b_{|\lambda \rho} \gtrless c_{|\lambda \rho}$. □

Polyendo-isochrony. Like endo-isochrony for endochronous processes, polyendo-isochrony is a property on the composition of polyendochronous processes. It ensures that a system composed of polyendochronous processes is equivalent in its synchronous and asynchronous composition.

Definition 24 (Polyendo-isochronous process). Let P_1 and P_2 be two processes defined respectively on V_1 and V_2. They are *polyendo-isochronous* iff P_1, P_2 and $(P_{1|V_1 \cap V_2} \mid P_{2|V_1 \cap V_2})$ are polyendochronous and $(P_{1|V_1 \cap V_2} \mid P_{2|V_1 \cap V_2})$ is acyclic.

Proposition 9. *If P_1 and P_2 are polyendo-isochronous, their asynchronous composition $P_1 \mid\mid P_2$ is flow-invariant.*

Proof. If P_1 and P_2 are polyendo-isochronous then they are polyendochronous, thus $P_1 \mid\mid P_1$ and $P_2 \mid\mid P_2$ are flow-invariant. Since flow-invariance is compositional, $P_1 \mid\mid P_2$ is flow-invariant. □

Polyhierarchy. Just like the hierarchical criteria of a compilable process may be used to test endochrony, we define a polyhierarchical property to test polyendochrony. Polyhierarchy is defined (recursively) as follows:

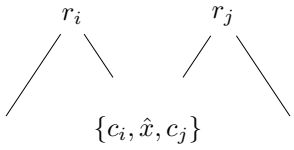
Definition 25 (Polyhierarchical process). If P is a compilable and hierarchical process, then it is polyhierarchical. If P and Q are polyhierarchical processes and $P \mid Q$ is compilable, then $P \mid Q$ is polyhierarchical.

Then, we have the following properties for polyhierarchical processes:

Proposition 10. *If P is a polyhierarchical process, then P is polyendochronous. If P and Q are polyhierarchical processes and $P \mid Q$ is compilable, then P and Q are polyendo-isochronous.*

Proof. By definition, a polyhierarchical process P is a composition of processes P_i which are compilable and hierarchical, thus endochronous. Since endochrony implies polyendochrony and polyendochrony is stable by composition, the composition P of processes P_i is polyendochronous.

Let P_i and P_j be two endochronous subprocesses from the composition $P \mid Q$, that share some variable x the clock of which is $\hat{x}$. The processes P_i and P_j have (respectively) r_i and r_j as roots and synchronize on $\hat{x}$ with (respectively) c_i and c_j. Since they are hierarchical, these clocks can be computed from the roots.



Since $P_i \mid P_j$ is compilable, the clocks c_i and c_j have disjunctive forms. Thus, the clock $\hat{x}$ can be calculated either, as c_i, only from clocks recursively dominated by r_i, or, as c_j, only from clocks recursively dominated by r_j. Thus, any reaction initiated in P_i that leads to $\hat{x}$ can locally decide to wait for a rendez-vous with a reaction of P_j containing $\hat{x}$. Since P_i and P_j are well-formed, the equation $\hat{x} = \hat{}0$ is excluded, and the reaction can thus occur. And since $P_i \mid P_j$ is acyclic, the rendez-vous of c_i and c_j cannot be locked by a deadlock.

For each such pair of processes P_i, P_j from $P \mid Q$, it is thus shown that this pair is non-blocking. The composition $P \mid Q$ is non-blocking.

Finally, this corresponds to the criteria of polyendo-isochrony. □

In [**?**], the authors proposed an operational method to characterize weakly endochronous processes. There technique is based on the construction of "generator sets" composed of partial reactions. These generator sets contain minimal synchronization patterns that characterize all possible reactions of a process. A process is weakly endochronous when its generator set is free of forced absence constraints. However, the proposed algorithms to compute generator sets involve an exploration of combinations of generators that can be very costly, with questionable scalability for realistic programs. To our knowledge, the method has not been implemented. On the contrary, the characterization of the polyendochronous processes that we propose, pertaining to the polyhierarchical property, is based on the calculation of the clock hierarchy, which has been proved as a very efficient and scalable technique [49]. It integrates smoothly within the Polychrony framework, which already uses the hierarchical criterion to characterize endochrony. Testing polyendochrony and generating code for polyendochronous processes (see Section 4) are now available in Polychrony.

A polyendochronous example. We consider a toy Signal example so that it can be easily demonstrated and understood. It is composed of specific simple devices: a *writer* and two instances of a *reader*. A realistic polyendochronous use case, namely the simulation model for a loosely time-triggered architecture (LTTA), could be obtained from more general versions of such devices, together with a model of *bus*: the LTTA [51, 52] is composed of a *writer*, a *bus* and a *reader*, each of them being paced by its own clock.

Here, our *writer* is paced by some Boolean clock *cw*; it delivers its input data *xw* on its output *yw* at the instants at which its clock *cw* is *tt* (other data values are lost):

```
process writer =
        ( ? integer xw; boolean cw;
          ! integer yw; )
  (| xw ^= cw
   | yw := xw when cw
   |);
```

The *reader* is paced by a Boolean clock *cr*. It reads data values on its input *xw* at the instants at which its clock *cr* is *tt*. At these instants, it delivers its

input data on its output *xr*; at the other instants of its clock, it delivers again the previous value (which has to be memorized):

```
process reader =
        ( ? integer yr; boolean cr;
          ! integer xr; )
   (| xr ^= cr
    | yr ^= [: cr]
    | xr := yr default (xr$ init 0)
    |);
```

Then, our example is the synchronous composition of the *writer* and the *reader* (in this case, two instances of the *reader*):

```
process write_read =
        (? integer xw; boolean cw, cr1, cr2;
         ! integer xr1, xr2; )
  (| yw := write(xw, cw)
   | xr1 := read(yw, cr1)
   | xr2 := read(yw, cr2)
   |)
  where integer yw;
    ...
end;
```

In this composition, the instants of the three Boolean clocks, *cw*, *cr1* and *cr2*, are left completely free, however, the instants at which they are *tt* are constrained to be strictly synchronous.

The following clock hierarchy is obtained by applying clock calculus:

$$\begin{array}{ccccc} \hat{}cw & & \hat{}cr1 & & \hat{}cr2 \\ | & & | & & | \\ [:cw] & \hat{}= & [:cr1] & \hat{}= & [:cr2] \end{array}$$

The clock hierarchy has three free clocks as roots, which are the clocks of the Booleans *cw*, *cr1* and *cr2*. However, clock constraints apply on subclocks of them. In some sense, the clock hierarchy appears as composed of three trees, each of them corresponding to an endochronous process, connected to the others at two rendezvous points. This observation justifies a possible code generation strategy for polyendochronous processes, which will be described in the next section.

4. From equations to process networks by model transformation

Different strategies may be applied to implement "endochronous process networks". These strategies, available in the Polychrony toolset, are de-

scribed in [53]: sequential code generation, clustered code generation with static scheduling, clustered code generation with dynamic scheduling, distributed code generation, as well as multithreaded, multi-core core generation [54, 55].

All these strategies consist of different source to source transformations. All of them proceed by analytic rewriting that refine a Signal process with an explicit control flow (obtained from the inference of hierarchical clock relations) and by a static schedule (obtained by the inference of causality relations). Additional user directives allow to map such processes on abstract architecture descriptions and interconnect them, before imperative code is finally generated.

In this section, we consider the latest of these strategies: to generate multi-threaded code for polyendochronous processes. We make the choice of focusing on the model transformations that come into play in its elaboration, rather than on the code generated that is generated, as the aim poly-endochrony is to exploit parallelism at its finest grain of modularity, rather than performing runtime optimization (using, e.g., WCET analysis and real-time schedule synthesis).

After the clock calculus, a program P is reorganized as a composition of processes, $P = (|\ P1 \mid P2 \mid \ \ldots \ \mid Pn\ |)$, each one structured around a clock tree. We have thus a composition of trees (the hierarchy is a set of trees). Each tree is dominated by a clock, which is a local root. Testing polyendochrony then consists in verifying that no clock formula associated with a given root is recursive (in this case, the program could be non-deterministic), and that clock formulas associated with roots do not use the clock difference operator ($\hat{}-$), i.e., there is no reference to the absence of a root clock (the absence of an input cannot be tested).

So we consider the program P as polyendochronous, with $Clk1 \mathrel{\hat{}=} Clk2$ as clock constraint, with $Clk1$ being a clock in the subtree corresponding to $P1$ and $Clk2$ a clock in the subtree corresponding to $P2$. Such a constraint induces a synchronization between two parts ($P1$, $P2$) of the program when $Clk1$ or $Clk2$ occurs. The principle of the code generation for polyendochronous processes is based on the existing distributed code generation [53], but with the additional resynchronization of parts of the application induced by the constraints on clocks ($Clk1$, $Clk2$) not placed in the same clock trees.

To prepare the code generation, the graph is modified by adding nodes to express the "rendezvous" between different parts of the application. A technique quite similar to the synchronization by semaphores is used. For that purpose, three *intrinsic* predefined processes are considered: *pK_Send*(), *pK_Receive*() and *pK_RendezVous*(). They are added as described below in the graph and explicit dependencies are also added. The *pK_Send* will allow to express to another part (say, $P2$) that the current part ($P1$) is at its point of rendezvous; the *pK_Receive* will specify that the current part is waiting for the rendezvous with another part; finally, the *pK_RendezVous* allows to manage the coherence of the logical instants between different parts.

For $Clk1$, the following definitions at clock $Clk1$ are added:

```
onClk1 :: (| n11 :: rdv1 := pK_Send()
```

```
              | n12 :: pK_Receive(rdv2)
              | n13 :: pK_RendezVous(rdv1, rdv2)
              | n11 --> n12 --> n13
              | rdv1 ^= rdv2 ^= Clk1
              | n13 --> N % for all nodes N assigned to Clk1
                          (excluding pK_Send/pK_Receive) %
              |)
```

Note that two additional syntactic notations, not used so far, appear here. These are first the notation x `-->` y, which expresses an explicit dependency between signals x and y; and the notation l :: P, which associates the label l with the process P (this label, viewed as a signal of type *event*, may be used in particular in synchronizations and explicit dependencies) [26].

Symmetrically, for *Clk*2, the following definitions at clock *Clk*2 are added:

```
onClk2 :: (| n21 :: rdv1 := pK_Send()
           | n22 :: pK_Receive(rdv2)
           | n23 :: pK_RendezVous(rdv2, rdv1)
           | n21 --> n22 --> n23
           | rdv1 ^= rdv2 ^= Clk2
           | n23 --> N % for all nodes N assigned to Clk2
                       (excluding pK_Send/pK_Receive) %
           |)
```

So the rewritten Signal program is now:

```
P' = (| (| P1 | onClk1 :: (| ... |) |)
     | (| P2 | onClk2 :: (| ... |) |)
     | P3 | ... | Pn |)
   = (| Q1 | Q2 | P3 | ... | Pn |)
```

Note that this writing induces that the signal *rdv*1 (resp., *rdv*2) becomes an output signal of $Q1$ (resp. $Q2$) and an input of $Q2$ (resp. $Q1$).

At this step, the technique used for distributed code generation in Polychrony is applied [53], considering that each hierarchy $Q1$, $Q2$, $P3$, ..., Pn will run on a specific processor. In this case, the purpose is mainly to partition the application and the processors will be virtual ones. The distribution technique requires in particular the application of a clustering on each partition. A cluster consists of all the nodes that depend on the same subset of the partition's inputs. Thus, the graph of a partition is a graph of clusters. A cluster may be executed as soon as its inputs are available, without any communication with the external world.

The Signal program is now defined by:

```
'' = (| (| Q11 | ... | Q1m1 |) % Clusters of Q1 %
     | (| Q21 | ... | Q2m2 |) % Clusters of Q2 %
     | ...
     | (| Pn1 | ... | Pmmn |) % Clusters of Pn %
     |)
```

This program can be easily rewritten in a new one by isolating the definition of the state variables for each virtual processor. This operation is required to update the state variables at the end of logical instants. The new Signal program is now:

```
P''' =
  (| (| labQ11' :: Q11' | ... | labQ1m1' :: Q1m1'
     | labQ1$ :: Q1$
     | labQ11' --> labQ1$ | ... | labQ1m1' --> labQ1$ |)
    % Clusters of Q1 with isolated state variables Q1$ %
   | (| labQ21' :: Q21' | ... | labQ2m2' :: Q2m2'
     | labQ2$ :: Q2$
     | labQ21' --> labQ2$ | ... | labQ2m2' --> labQ2$ |)
    % Clusters of Q2 with isolated state variables Q2$ %
   | ...
   | (| labPn1' :: Pn1' | ... | labPnmn' :: Pnmn'
     | labPn$ :: Pn$
     | labPn1' --> labPn$ | ... | labPnmn' --> labPn$ |)
    % Clusters of Pn with isolated state variables Pn$ %
   |)
```

Then, a "step manager" (represented in the example below by $Q1iter$, labeled by $labQ1iter$) is added to each partition. For example, for

```
Q1 = (| labQ11' :: Q11' | ... | labQ1m1' :: Q1m1'
      | labQ1$ :: Q1$
      | labQ11' --> labQ1$ | ... | labQ1m1' --> labQ1$ |)
```

it is completed in:

```
Q1' =
(| labQ11' :: Q11' | ... | labQ1m1' :: Q1m1'
 | labQ1$ :: Q1$
 | labQ11' --> labQ1$ | ... | labQ1m1' --> labQ1$
 | labQ1iter :: Q1iter
 | (| labQ1$ --> labQ1iter
    | labQ11' --> labQ1iter | ... | labQ1m1' --> labQ1iter
    | labQ1iter --> labX
       % for all labX in {labQ11', ..., labQ1m1'}
          and X without predecessor in the graph of the clusters %
    |)
 |)
```

The code generation of this partition consists in the definition of several tasks:

- one task per cluster (i.e., $Q11'$, ..., $Q1m1'$),
- one task per input/output of the partition,

- one task for the cluster of state variables (i.e., $Q1\$$),
- one task that manages the steps ($Q1iter$).

Synchronization between these tasks is obtained by semaphores (one semaphore per task). Such a task has the following scheme:

```
WAIT;
code;
SIGNAL
```

where:

- "WAIT" ia a set of "wait" instructions on the semaphore associated with the task. The number of "wait" is equal to the number of predecessors in the graph of the clusters (in the considered partition).
- "code" is the code of the cluster or the code of the read/write for the inputs/outputs; it is empty for the task that manages the steps.
- "SIGNAL" is a set of "signal" instructions on semaphores. The number of "signal" is equal to the number of successors in the graph of the clusters. For example, consider a task Tj that implements the cluster Cj, with Sj as associated semaphore. If Cj depends on the cluster Ci, implemented by the task Ti, the instruction "signal(Sj)" will be generated at the end of the task Ti.

The iterate task ($Q1iter$) waits for the end of the other tasks (end of a logical instant) and signals the beginning of a new instant to the clusters without predecessor. In this way, the presented mechanism ensures the dynamic scheduling of tasks. Moreover, to ensure a correct resynchronization, this code generation has to be completed specifically for the partitions involved in the clock constraint. For such a partition ($Q1'$ for example), the following actions are applied:

- a local step counter (which counts logical instants) is introduced to resynchronize the partition $Q1'$ with the other one ($Q2'$);
- the code associated with the $pK\_Send()$ intrinsic process, for example $rdv1 := pk\_Send()$, consists in the assigment of the value of the local step counter to $rdv1$;
- the $pK\_receive()$ intrinsic process does not produce any code;
- and the $pK\_RendezVous(rdv1, rdv2)$ produces the generation of ($rdv2 - rdv1$) "$\perp$ instants" if the $rdv1$ is lower than $rdv2$, where $rdv2$ is sent by the other partition ($Q2'$) and its value is the local step counter of this partition (indeed, in this case, the partition $Q2'$ has taken some advance on the partition $Q1'$).

This method may be easily generalized for n constraints on a same clock. For that, the *pK_RendezVous* node has a variable number of inputs depending on the number of constraints on the same clock. Typically, a constraint on three clocks ($Clk1 \hat{=} Clk2 \hat{=} Clk3$) placed in different clock trees induces the definition of the following nodes:

- $n13 ::$ $pK\_RendezVous(rdv1, rdv2, rdv3)$ in the hierarchy of $P1$,
- $n23 ::$ $pK\_RendezVous(rdv2, rdv1, rdv3)$ in the hierarchy of $P2$,
- and $n33 ::$ $pK\_RendezVous(rdv3, rdv1, rdv2)$ in the hierarchy of $P3$.

For example, $pK\_RendezVous(rdv1, rdv2, rdv3)$ in the hierarchy of $P1$ produces the generation of $(m - rdv1)$ "$\perp$ instants" if the $rdv1$ is lower than $m = max(rdv2, \ldots, rdvn)$, where the values $rdv2$, ..., $rdvn$ are sent by the other partitions.

5. KPN are (extended) Signal programs

In Section 3, we have considered the question "are Signal programs KPNs?" In this section, we consider the more or less inverse interrogation: "how can KPNs be viewed as Signal programs?" In particular, which extensions to Signal could be required?

As a preliminary remark, it should be noticed that an actor in a KPN can be defined recursively. This is not the case of Signal programs (though bounded recursion would not be conflicting with the principles of the language).

We first give the principle of the description of a bounded FIFO in Signal (Section 5.1). Then, in Section 5.2, we propose an extension of Signal to handle asynchronous communications and in Section 5.3, we revisit asynchronous composition. Associated with counters of events, this allows to express synchronized Signal programs from KPNs (Section 5.4).

5.1. Bounded FIFO

Recall that, in the implementation model of a KPN, actors communicate via FIFO channels. If the specification of general unbounded FIFOs is out of the scope of the Signal language (the purpose of which is mainly the description of real-time safety-critical applications), on the other hand, *bounded* FIFOs can be specified as Signal programs.

The following program partly describes a FIFO buffer of size *fifo_size*:

```
process FIFO =
  { type message_type; integer fifo_size;
  ( ? message_type mess_in;
    ! message_type mess_out; )
  (| index := ((prev_index + 1) when ^mess_in) default
              ((prev_index - 1) when ^mess_out) default
              prev_index
```

```
    | prev_index := index$ init 0
    | index ^= mess_in ^+ mess_out
    | OK_write := prev_index < fifo_size
    | OK_read := prev_index > 0
    | ...
    |)
  where ...
end;
```

The signal *index* represents the index of the first free cell in the buffer. Note that in this simple specification, it is considered that a write (signal *mess_in*) and a read (signal *mess_out*) actions in the buffer cannot be simultaneous. In addition, this partial specification represents only the control part of the FIFO (the full specification uses the Signal sliding window operator—derived from the delay—which is not presented in this article).

The Boolean signals *OK_write* and *OK_read* have the value *tt* when it is possible to write (respectively, to read) a new data in the FIFO. To ensure that the FIFO is safe, the following constraints are added (as assertions) to the process: *mess_in* $\hat{}<$ (*when OK_write*) and *mess_out* $\hat{}<$ (*when OK_read*).

5.2. Equations in extended Signal

We have seen that Signal equations, written $x := y$, relate extended streams, which are sequences of values *and* meaningful absence. The equation $x := y$ expresses *synchronized flow equality*. There is a *synchronous communication* between y and x.

Kahn Process Networks describe flow functions. The absence is not explicitly considered as a value. An extension of Signal as flow functions would mean the introduction of a new, idealized, class of equations, written $x :- y$ to denote the equality between the considered streams as pure flows (that is, infinite sequences of values, filtering or discarding the absence of values).

By analogy with flow-equivalence defined in Section 2.3, we can say that x and y are flow-equivalent through $x :- y$. Since both signals hold the same sequences of values, with possibly any number of absence value between these values, the equation denotes an idealized *asynchronous communication* between y and x.

5.3. Back to asynchronous composition

The equation $x :- y$ represents the behavior of an unbounded FIFO buffer. By using it, it is possible to express the asynchronous composition $P \mid\mid Q$ of processes P and Q (see Section 2.4) as their synchronous composition with unbounded FIFOs (i.e. as it becomes continuous). Let $(x_i)_i$ be the signals shared by P and Q. Then, their asynchronous composition is equal to

$$P \mid\mid Q = P[(x_{ip}/x_i)_i] \mid (\ x_{ip} :- x_{iq} \ )_i \mid Q[(x_{iq}/x_i)_i]$$

where the notation $P[(x_{ip}/x_i)_i]$ represents the substitution of variables shared by P and Q. KPNs can hence be expressed as such Signal programs "extended" with idealized unbounded FIFOs.

5.4. Back to synchronized flows

Let us associate each signal x with a counter $\#x$ of its valued events. This counter can be defined by the signal nx in the following process:

$$(|\ nx := ((znx + 1)\ when\ \hat{}x)\ default\ znx \mid znx := nx\$\ init\ 0\ |)/znx$$

A bounded FIFO between y and x, with n slots, can then be defined as:

$$(|\ x :- y \mid (|\ counter := \#y - \#x \mid counter\ \hat{}=\ when\ (counter <= n)\ |)\ |)$$

The equation $counter\ \hat{}=\ when\ (counter <= n)$, as an assertion, constrains the signal *counter* to be present only when its value is less than or equal to n. Then, a "synchronized program" can be expressed as a Signal program with unbounded FIFOs (a KPN), composed with a set of such bound constraints.

6. Conclusion

In this paper, we examined the polychronous model of the synchronized dataflow language Signal with respect to the well-known model of Kahn Process Networks. In particular, we tried to answer two questions:

1/ When are Signal programs Kahn Process Networks?

2/ When can Kahn Process Networks be seen as Signal programs?

To answer the first question, we have shown the somewhat unexpected role of the Signal clock calculus. To answer the second one, we have proposed a small syntactic extension of the Signal language that makes the idealized, mathematical, notion of flow-equivalence syntactically explicit.

The "takeaway" of this comparison is to invite the reader enrich implementation techniques used by one's community with analytic techniques used by the other's. From a theoretical point of view, the extension of Signal proposed in Section 5 could for instance lead one revisit the polychronous model. Relations on flows, together with event counters, could be used to express unbounded asynchronous or weakly synchronous communication (e.g. loosely time-triggered systems), and/or to redefine synchronous communication from more primitive operations. From a practical point of view, we have described program transformation techniques for the special class of networks called "polyendochronous processes" in the Polychrony toolset.

References

[1] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., 1985.

[2] R. Milner, A Calculus of Communicating Systems, Springer-Verlag, 1980.

[3] J. A. Bergstra, J. W. Klop, Process algebra for synchronous communication, Information and Control 60 (1–3) (1984) 109–137.

[4] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes (parts I and II), Information and Computation 100 (1) (1992) 1–77.

[5] E. A. Lee, T. M. Parks, Dataflow process networks, Proceedings of the IEEE 83 (5) (1995) 773–801.

[6] G. Kahn, The semantics of a simple language for parallel programming, in: J. L. Rosenfeld (Ed.), Information Processing '74: Proceedings of the IFIP Congress, North-Holland, New York, NY, 1974, pp. 471–475.

[7] A. Benveniste, G. Berry, The synchronous approach to reactive and real-time systems, Proceedings of the IEEE 79 (9) (1991) 1270–1282.

[8] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone, The synchronous languages twelve years later, Proceedings of the IEEE, Special issue on Modeling and Design of Embedded Systems 91 (1) (2003).

[9] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, Programming real-time applications with SIGNAL, Proceedings of the IEEE 79 (9) (1991) 1321–1336.

[10] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann, Polychrony for system design, Journal for Circuits, Systems and Computers 12 (3) (2003) 261–304.

[11] A. Gamatié, Designing Embedded Systems with the SIGNAL Programming Language, Springer, 2009.

[12] J. A. Bergstra, J. W. Klop, J. V. Tucker, Process algebra with asynchronous communication mechanisms, in: Seminar on Concurrency, Carnegie-Mellon University, Springer-Verlag, 1985, pp. 76–95.

[13] R. Milner, Calculi for synchrony and asynchrony, Theoretical Computer Science 25 (3) (1983) 267–310.

[14] P. Le Guernic, T. Gautier, Data-flow to von Neumann: the SIGNAL approach, in: J.-L. Gaudiot, L. Bic (Eds.), Advanced Topics in Data-Flow Computing, 1991, pp. 413–438.

[15] P. Caspi, Clocks in dataflow languages, Theoretical Computer Science 94 (1) (1992) 125–140.

[16] P. Caspi, A. Girault, Execution of distributed reactive systems, in: S. Haridi, K. A. M. Ali, P. Magnusson (Eds.), Euro-Par '95 Parallel Processing, First International Euro-Par Conference, Stockholm, Sweden, August 29-31, 1995, Proceedings, Vol. 966 of Lecture Notes in Computer Science, Springer, 1995, pp. 15–26. doi:10.1007/BFb0020452.
URL https://doi.org/10.1007/BFb0020452

[17] G. Berry, E. Sentovich, An implementation of constructive synchronous programs in POLIS, Formal Methods Syst. Des. 17 (2) (2000) 135–161. doi:10.1023/A:1008796718837.
URL https://doi.org/10.1023/A:1008796718837

[18] A. Benveniste, B. Caillaud, P. Le Guernic, Compositionality in dataflow synchronous languages: Specification and distributed code generation, Information and Computation 163 (1) (2000) 125–171.

[19] J. Talpin, D. Potop-Butucaru, J. Ouy, B. Caillaud, From multi-clocked synchronous processes to latency-insensitive modules, in: W. H. Wolf (Ed.), EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings, ACM, 2005, pp. 282–285. doi:10.1145/1086228.1086279.
URL https://doi.org/10.1145/1086228.1086279

[20] D. Potop-Butucaru, B. Caillaud, A. Benveniste, Concurrency in synchronous systems, Formal Methods in System Design 28 (2) (2006) 111–130.

[21] D. Potop-Butucaru, B. Caillaud, Correct-by-construction asynchronous implementation of modular synchronous specifications, Fundamenta Informaticae 78 (1) (2007) 131–159.

[22] D. Potop-Butucaru, R. de Simone, Y. Sorel, J.-P. Talpin, From concurrent multiclock programs to deterministic asynchronous implementations, in: 2009 9th International Conference on Application of Concurrency to System Design (ACSD 2009), IEEE, Augsburg, Germany, 2009, pp. 42–51.

[23] L. P. Carloni, K. L. McMillan, A. L. Sangiovanni-Vincentelli, Latency insensitive protocols, in: N. Halbwachs, D. A. Peled (Eds.), Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings, Vol. 1633 of Lecture Notes in Computer Science, Springer, 1999, pp. 123–133. doi:10.1007/3-540-48683-6_13.
URL https://doi.org/10.1007/3-540-48683-6_13

[24] J. Ouy, Génération de code asynchrone dans un environnement polychrone pour la production de systèmes GALS, Ph.D. thesis, Université de Rennes 1, France (Jan. 2008).

[25] J.-P. Talpin, J. Ouy, T. Gautier, L. Besnard, P. Le Guernic, Compositional design of isochronous systems, Science of Computer Programming 77 (2) (2012) 113–128.

[26] L. Besnard, T. Gautier, P. Le Guernic, SIGNAL V4-INRIA version: Reference Manual (2020).
URL http://polychrony.inria.fr/document/V4_def.pdf

[27] M. Geilen, T. Basten, Kahn process networks and a reactive extension, in: S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, J. Takala (Eds.), Handbook of Signal Processing Systems, Springer, 2010, pp. 967–1006.

[28] J. D. Brock, W. B. Ackerman, Scenarios: A model of non-determinate computation, in: Proceedings of the International Colloquium on Formalization of Programming Concepts, Springer-Verlag, London, UK, 1981, pp. 252–259.

[29] P. Panangaden, The expressive power of indeterminate primitives in asynchronous computation, in: Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, London, UK, 1995, pp. 124–150.

[30] P. Panangaden, V. Shanbhogue, The expressive power of indeterminate dataflow primitives, Information and Computation 98 (1) (1992) 99–131.

[31] P. Raymond, N. Halbwachs, P. Caspi, D. Pilaud, The synchronous dataflow programming language Lustre, Proceedings of the IEEE 79 (9) (1991) 1305–1320.

[32] T. P. Amagbegnon, L. Besnard, P. Le Guernic, Implementation of the dataflow synchronous language Signal, in: Programming Languages Design and Implementation, ACM Press, La Jolla, California, 1995, pp. 163–173.

[33] P. Caspi, M. Pouzet, Synchronous Kahn networks, SIGPLAN Notices 31 (6) (1996) 226–238.

[34] P. Wadler, Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time, in: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming, ACM, New York, NY, USA, 1984, pp. 45–52.

[35] E. A. Lee, D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Transactions on Computers 36 (1) (1987) 24–35.

[36] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, H. Zheng, Overview of the Ptolemy project, Tech. rep., University of California, Berkeley, Berkeley, CA, 94720, USA (July 2003).
URL `http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/overview/`

[37] C. Ptolemaeus (Ed.), System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014.

[38] P. Caspi, A. Benveniste, R. Lublinerman, S. Tripakis, Actors without directors: A Kahnian view of heterogeneous systems, in: HSCC '09: Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 46–60.

[39] E. A. Lee, A. Sangiovanni-Vincentelli, A framework for comparing models of computation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17 (1998) 1217–1229.

[40] X. Liu, E. A. Lee, CPO semantics of timed interactive actor networks, Theoretical Computer Science 409 (1) (2008) 110–125.

[41] J. T. Buck, E. A. Lee, Scheduling dynamic dataflow graphs with bounded memory using the token flow model, in: Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 1, IEEE Computer Society, 1993, pp. 429–432.

[42] T. M. Parks, Bounded scheduling of process networks, Ph.D. thesis, University of California at Berkeley, Berkeley, CA, USA (1995).

[43] R. S. Stevens, M. Wan, P. Laramie, T. M. Parks, E. A. Lee, Implementation of process networks in Java, Tech. Rep. UCB/ERL M97/84, EECS Department, University of California, Berkeley (1997).
URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1997/3335.html`

[44] M. Goel, Process networks in Ptolemy II, Tech. Rep. UCB/ERL M98/69, EECS Department, University of California, Berkeley (1998).
URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1998/3542.html`

[45] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, G. Essink, YAPI: application modeling for signal processing systems, in: DAC '00: Proceedings of the 37th Annual Design Automation Conference, ACM, New York, NY, USA, 2000, pp. 402–405.

[46] M. Geilen, T. Basten, Requirements on the execution of Kahn process networks, in: Proc. of the 12th European Symposium on Programming, ESOP 2003, Springer Verlag, 2003, pp. 319–334.

[47] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, M. Pouzet, N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems, in: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 2006, pp. 180–193.

[48] A. Cohen, L. Mandel, F. Plateau, M. Pouzet, Abstraction of clocks in synchronous data-flow systems, in: APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 237–254.

[49] T. P. Amagbegnon, L. Besnard, P. Le Guernic, Arborescent canonical form of boolean expressions, Research Report RR-2290, INRIA (1994).

[50] L. Besnard, Compilation de SIGNAL : horloges, dépendances, environnement, Ph.D. thesis, Université de Rennes 1, France (Sep. 1992).

[51] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, S. Tripakis, A protocol for loosely time-triggered architectures, in: A. Sangiovanni-Vincentelli, J. Sifakis (Eds.), Embedded Software, Springer, 2002, pp. 252–265.

[52] Y. Glouche, T. Gautier, P. Le Guernic, J.-P. Talpin, A module language for typing SIGNAL programs by contracts, in: S. K. Shukla, J.-P. Talpin (Eds.), Synthesis of Embedded Software, Springer, 2010, pp. 147–171.

[53] L. Besnard, T. Gautier, P. Le Guernic, J.-P. Talpin, Compilation of polychronous data flow equations, in: S. K. Shukla, J.-P. Talpin (Eds.), Synthesis of Embedded Software, Springer, 2010, pp. 1–40.

[54] Z. Yang, J.-P. Bodeveix, M. Filali, K. Hu, Y. Zhao, D. Ma, Towards a verified compiler prototype for the synchronous language SIGNAL, Frontiers of Computer Science 10 (1) (2016) 37–53.

[55] Z. Yang, S. Yuan, J.-P. Bodeveix, M. Filali, T. Wang, Y. Zhou, Multi-task Ada code generation from synchronous dataflow programs on multi-core: Approach and industrial study, Science of Computer Programming 207 (2021).