Abstract

The design of high-performance stream-processing systems is a fast growing domain, driven by markets such as high-end TV, gaming, 3D animation and medical imaging. It is also a surprisingly demanding task, with respect to the algorithmic and conceptual simplicity of streaming applications. It needs the close cooperation between numerical analysts, parallel programming experts, real-time control experts and computer architects, and incurs a very high level of quality assurance and optimization.

In search for improved productivity, we propose a programming model and language dedicated to high-performance stream processing. This language builds on the synchronous programming model and on domain knowledge — the periodic evolution of streams — to allow,
system with multiple input/output associated with different real-time clocks (e.g., video streaming). It is thus necessary to compose independently scheduled processes. Kahn Process Networks (KPN) [18] can accommodate for such a composition, compensating for the local asynchrony through unbounded blocking FIFO buffers. But allowing a global synchronous execution imposes additional constraints on the composition. We introduce the concept of n-synchronous clocks to formalize these concepts and constraints. This concept describes naturally the semantics of KPN components with multiple periodic clocks satisfying a flow preserving property of FIFO buffers. This principle is currently restricted to periodic buffers. But allowing a global synchronous execution imposes additional constraints on the composition. We introduce the concept of n-synchronous clocks to formalize these concepts and constraints. This concept describes naturally the semantics of KPN components with multiple periodic clocks satisfying a flow preserving property of FIFO buffers.

Main Contributions More technically, we define a relaxed clock-equivalence principle, called n-synchrony. A given clock ck₁ is n-synchronizable with another clock ck₂ if there exists a data-flow (causality) preserving way of making ck₁ synchronous with ck₂ applying a constant delay to ck₂ and inserting an intermediate size n FIFO buffer. This principle is currently restricted to periodic clocks defined as periodic infinite binary words. This is different and independent from retiming [20], since neither ck₁ nor ck₂ are modified (besides the optional insertion of a constant delay); schedule choices associated with ck₁ and ck₂ are not impacted by the synchronization process.

We also define a relaxed synchronous functional programming language whose clock calculus accepts n-synchronous composition of operators. To this end, a type system underlying a strictly synchronous model through the presentation of the downscaler, enforcing bounded resource requirements and real-time execution. Yet, we show that existing synchronous languages make such an implementation tedious and error-prone.

Paper Outline The structure of the paper is the following. In Section 2, we motivate the n-synchronous model through the presentation of a simple high-performance video application. Section 3 formalizes the concepts of periodic clocks and synchronizability. Section 4 is our main contribution: starting from a core synchronous language a la LUSTRE, it presents an associated calculus on periodic clocks and extends this calculus to combine streams with n-synchronizable clocks. Section 5 describes the semantics of n-synchronous clock calculus is extended with two subtyping rules. Type inference follows an ad-hoc but complete procedure.

We show that every n-synchronous program can be transformed into a synchronous one (0-synchronous), replacing bounded buffers by some synchronous code.

2. Motivation

Although this work may contribute to the design of a wide range of embedded systems, we are primarily driven by video stream processing for high-definition TV [16]. The main algorithms deal with picture scaling, picture composition (picture-in-picture) and quality enhancement (including picture rate up-conversions; converting the frame rate of the displayed video, de-interlacing flat panel displays, sharpness improvement, color enhancement, etc.). Processing requires considerable resources and involves a variety of pipelined algorithms on multidimensional streams.

These applications involve a set of scalers that resize images in real-time. Our running example is a classical downsampler [9], depicted in Figure 1. It converts a high definition (HD) video signal, 1920 × 1080 pixels per frame, into a standard definition (SD) output for TV screen, that is 720 × 480:

1. A horizontal filter, hf, reduces the number of pixels in a line from 1920 down to 720 by interpolating packets of 6 pixels.
2. A reordering module, reorder, stores 6 lines of 720 pixels.
3. A vertical filter, vf, reduces the number of lines in a frame from 1080 down to 480 by interpolating packets of 6 pixels.

The processing of a given frame involves a constant number of operations on this frame only. A design tool is thus expected to automatically produce an efficient code for an embedded architecture, to check that real-time constraints are met, and to optimize the memory footprint of intermediate data and of the control code. The embedded system designer is looking for a programming language that offers precisely these features, and more precisely, which statically guarantees four important properties:

1. a proof that, according to worst-case execution time hypotheses, the frame and pixel rate will be sustained;
2. an evaluation of the delay introduced by the downscaler in the video processing chain, i.e., the delay before the output process starts receiving pixels;
3. a proof that the system has bounded memory requirements;
4. an evaluation of memory requirements, to store data within the processes, and to buffer the stream produced by the vertical filter in front of the output process.

In theory, synchronous languages are well suited to the implementation of the downscaler, enforcing bounded resource requirements and real-time execution. Yet, we show that existing synchronous languages make such an implementation tedious and error-prone.

2.1 The Need to Capture Periodic Execution

Technically, the scaling algorithm produces its t-th output (oₜ) by interpolating 6 consecutive pixels (pᵢ) weighted by coefficients given in a predetermined matrix (example of a 64 phases, 6-taps polyphase filter [9]):

\[ oₜ = \sum_{k=0}^{5} P(r \times 1920/720) \times k \mod 64. \]

let clock c = ok where rec
\ntcnt = 1 fby (if (cnt = 8) then 1 else cnt + 1)
and ok = (cnt = 1) or (cnt = 3) or (cnt = 6)

let node hf p = o where rec
\no2 = 0 fby p and o3 = 0 fby o2 and o4 = 0 fby o3
and o5 = 0 fby o4 and o6 = 0 fby o5
and o = f (p,o2,o3,o4,o5,o6) when c

val hf : int -> int
val hf :: 'a -> 'a on c

Figure 2. Synchronous implementation of hf

Such filtering functions can easily be programmed in a strictly synchronous data-flow language such as LUSTRE or LUCID SYCHRONIX. Figure 2 shows a first version of the horizontal filter implemented in LUCID SYCHRONIX.

1 Here we only consider the active pixels for the ATSC or BS-Digital High Definition standards.
At every clock tick, the \( \mathsf{hf} \) function computes the interpolation of six consecutive pixels of the input \( p \) (0\, \text{E} \, p \) stands for the previous value of \( p \) initialised with value 0). The implementation of \( \mathsf{f} \) is out of the scope of this paper; we will assume it sums its 6 arguments. The horizontal filter must match the production of 3 pixels for 8 input pixels. Moreover, the signal processing algorithm defines precisely the time when every pixel is emitted: the \( t \)-th output appears at the \( t \times 1920/720 \)-th input. It can be factored in a periodic behavior of size 8, introducing an auxiliary boolean stream \( c \) used as a clock to sample the output of the horizontal filter. The clock construction identifies syntactically these particular boolean streams. Here is a possible execution diagram.

\[
\begin{array}{cccccccc}
\alpha & \alpha & \alpha & \alpha & \alpha & \alpha & \alpha & \alpha \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
\mathsf{hf} & \mathsf{hf} & \mathsf{hf} & \mathsf{hf} & \mathsf{hf} & \mathsf{hf} & \mathsf{hf} & \mathsf{hf} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

In the synchronous data-flow model, each variable/expressions is characterized both by its stream of values and by its clock, relative to a global clock, called the base clock of the system. The clock of any expression \( e \) is an infinite boolean stream where \( \mathsf{false} \) stands for the absence and \( \mathsf{true} \) for the presence. E.g., if \( x \) is an integer stream variable, then \( x \times 1 \) and \( x \times 2 \) have the same clock. A synchronous process transforms an input clock into an output clock. This transformation is encoded in the process clock signature or clock type. Clocks signatures are relative to some clock variables. E.g., the clock signature of \( \mathsf{hf} \) is \( \forall \alpha. \alpha \to \alpha \) on \( c \) (printed \( \mathsf{ta} \rightarrow \mathsf{ta} \) on \( c \)) meaning that for any clock \( \alpha \), if \( p \) has clock \( \alpha \), then the output is on a subclock \( \alpha \) on \( c \) defined by the instant where the boolean condition \( c \) is \( \mathsf{true} \).

2.2 The Need for a Relaxed Approach

Real-time constraints on the filters are deduced from the frame rate: the input and output processes enforce that frames are sent and received at 30Hz. This means that HD pixels arrive at 30 \( \times 1920 \times 1080 = 62,208,000 \)Hz — called the HD pixel clock — and SD pixels at 30 \( \times 720 \times 480 = 10,368,000 \)Hz — called the SD pixel clock — i.e., 6 times slower. From these numbers, the designer would like to know that the delay before seeing the first output pixel is actually 12000 cycles of the HD pixel clock, i.e., 192.9156\( \mu \)s, and that the minimal size of the buffer between the vertical filter and output process is 880 pixels.

Synchronous languages typically offer such guarantees and static evaluations by forcing the programmer to make explicit the synchronous execution of the application. Nevertheless, the use of any synchronous language requires the designer to explicitly implement a synchronous code to buffer the output pixels at the proper output rate and nothing helps him/her to automatically compute the values 12000 and 880. Unfortunately, pixels are produced by the downscaler following a periodic but complex event clock. The synchronous code for the buffer handles the storage of each pending write from the vertical filter into a dedicated register, until the time for the output process to fetch this pixel is reached. Forcing the programmer to provide the synchronous buffer code is thus tedious and breaks modular composition. This scheme is even more complex if we include blanking periods [16].

In the following, we design a language that makes the computation of process latencies and buffer sizes automatic, using explicit periodic clocks.

3. Ultimately Periodic Clocks

This section introduces the formalism for reasoning about periodic clocks of infinite data streams.

3.1 Definitions and Notations

Infinite binary words are words of \( (0 + 1)^\omega \). For the sake of simplicity, we will assume thereafter that every infinite binary word has an infinite number of 1s.

We are mostly interested in a subset of these words, called infinite ultimately periodic binary words or simply infinite periodic binary words, defined by the following grammar:

\[
\begin{align*}
\mathsf{w} & ::= u(v) \\
\mathsf{u} & ::= 1 | 0 | 1.\mathsf{u} | 1.\mathsf{v} \\
\mathsf{v} & ::= 0 | 1 | 0.\mathsf{v} | 1.\mathsf{v}
\end{align*}
\]

where \( (v) = \lim_{n \to \infty} v^n \) denotes the infinite repetition of period \( v \), and \( u \) is a prefix of \( w \). Let \( \mathbb{Q}_2 \) denote the set of infinite periodic binary words; it coincides with the set of rational diadic numbers [25]. Since we always consider infinite periodic binary words with an infinite number of 1s, the period \( v \) contains at least one 1. This corresponds to removing the integer numbers from \( \mathbb{Q}_2 \) and considering only \( \mathbb{Q}_2 - \overline{\mathbb{N}} \).

Let \( |w| \) denote the length of \( w \). Let \( |w|_1 \) denote the number of 1s in \( w \) and \( |w|_0 \) the number of 0s in \( w \). Let \( w[n] \) denote the \( n \)-th letter of \( w \) for \( n \in \mathbb{N} \) and \( w[1..n] \) the prefix of length \( n \) of \( w \).

There are an infinite number of representations for an infinite periodic binary word. Indeed, \((0101)\) is equal to \((0110)\) and to \((0111)\). Fortunately, there exists a normal representation: it is the unique representation of the form \( u(v) \) with the shortest prefix \( u \) and with the shortest period.

Let \( |w|_p \) denote the position of the \( p \)-th 1 in \( w \). We have \(|1.w|_1 = 1, |1.w|_p = |w|_p-1 + 1 \) if \( p > 1 \), and \(|0.w|_p = |w|_p + 1 \). Finally, let us define the precedence relation \( \preceq \) by

\[
w_1 \preceq w_2 \iff \forall p \geq 1, |w_1|_p \leq |w_2|_p.
\]

E.g., \((10) \preceq (01) \preceq (001) \preceq (0011)\). This relation is a partial order on infinite binary words. It abstracts the causality relation on stream computations, e.g., to check that outputs are produced before consumers request them as inputs.

We can also define the upper bound \( w \cup w' \) and lower bound \( w \cap w' \) of two infinite binary words with

\[
\begin{align*}
\forall p \geq 1, |w \cup w'|_p &= \max(|w|_p, |w'|_p) \\
\forall p \geq 1, |w \cap w'|_p &= \min(|w|_p, |w'|_p).
\end{align*}
\]

E.g., \((10) \cup (01) = (01) \) and \((10) \cap (01) = (10)\). Finally, \((010) \cup (0101) = (0101) \) and \((01001) \cap (01100) = (10)\).

Proposition 1. The set \((0 + 1)^\omega, \preceq, \cup, \cap, \_ \equiv (1), \_ \equiv (0) \) is a complete lattice.

Notice \( \_ \) is indeed \( (0) \) since \([0]_p = \infty \) for all \( p > 0 \).

Eventually, the following remark allows most operations on infinite periodic binary words to be computed on finite words.

Remark 1. Considering two infinite periodic binary words, \( w = u(v) \) and \( w' = u'(v') \), one may transform these expressions into equivalent representatives \( a(b) \) and \( a'(b') \) satisfying one of the following conditions.

\[\text{Yet the restriction of this lattice to } \mathbb{Q}_2 \text{ is not complete, neither upwards nor downwards, even within } \mathbb{Q}_2 - \overline{\mathbb{N}}.\]
1. One may choose a, b, and b' with |a| = |a'| = \max(|a|, |a'|) and |b| = |b'| = \text{lcm}(|v|, |v'|) where lcm stands for least common multiple. Indeed, assuming |a| ≤ |a'|, p = |a' - |a| and n = \text{lcm}(|v|, |v'|); w = u.a[1]...v[p].v[p + 1]...v[p + |v'|]n(|v'|) and w' = u'.v(r/(n|v'|)). E.g., words 010(001100) and 01001(10) can be rewritten into 010010(100010) and 010010(101010).

2. Likewise, one may obtain prefixes and suffixes with the same number of Is: w = a(b) and w' = a'(b') with |a| = |a'| = \max(|a|, |a'|) and |b| = |b'| = \text{lcm}(|v|, |v'|). Indeed, suppose |a| ≤ |a'|, |v| ≤ |v'|, p = |a' - |a|, r = |v'|, and n = \text{lcm}(|v|, |v'|): w = u.a[1]...v[p].v[p + 1]...v[p + |v'|]n(|v'|) and w' = u'.v(r/(n|v'|)). E.g., the pair of words 010(001100) and 01001(10) become 010010(100010) and 010010(101010).

3. Finally, one may write w = a(b) and w' = a'(b') with |a| = |a'| and |b| = |b'|. Indeed, suppose |a| ≤ |a'|, |v| ≤ |v'|, p = |a' - |a|, r = |v'|, and n = \text{lcm}(|v|, |v'|): w = u.a[1]...v[p].v[p + 1]...v[p + |v'|]n(|v'|) and w' = u'.v(r/(n|v'|)). E.g., the pair of words 010(001100) and 01001(10) can be rewritten into 010010(100010) and 010010(101010).

3.2 Clock Sampling and Periodic Clocks

A clock for infinite streams can be an infinite binary word or a composition of those, as defined by the following grammar:

\[
c ::= w \mid c \cdot w, \quad w \in \{0, 1\}^\omega.
\]

If c is a clock and w is an infinite binary word, then c \cdot w denotes a subsampled clock of c, where w is itself set on clock c. In other words, c \cdot w is the clock obtained in advancing in clock w at the pace of clock c. E.g., (01) \cdot (101) = (0101011) \cdot (101) = (01001).

\[
\begin{array}{c|cccccccc}
  c & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \hline
  w & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \hline
  c \cdot w & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}
\]

Formally, on is inductively defined as follows:

\[
0.c \cdot w' = 0.\text{w}(w') \quad 1.c \cdot 0.w' = 0.\text{w}(w') \quad 1.c \cdot 1.w' = 1.\text{w}(w')
\]

Clearly, the \(\cdot\) operator is not commutative.

**Proposition 2.** Given two infinite binary words w and w', the infinite binary word w \cdot w' satisfies the equation \[\text{w}(w \cdot w')_p = \text{w}(w)_p \cdot \text{w}(w')_p\] for all \(p \geq 1\).

**Proof.** This is proven by induction, observing that w' is traversed at the rate of 1s in w. \[\text{w}(w \cdot w')_1\] is associated with the q-th 1 of w such that q is the rank of the first 1 in w', i.e., q = |w'|. Assuming the equation is true for p, the same argument proves that \[\text{w}(w \cdot w')_p+1 = \text{w}(w)_p \cdot \text{w}(w')_p\] where q is the distance to the next 1 in w', i.e., q = |w'| + 1 - |w'|. This concludes the proof.

3.3 Synchronizability

Motivated by the downsampling example, we introduce an equivalence relation to characterize the concept of resynchronization of infinite binary words (not necessarily periodic).

**Definition 1** (synchronizable words). We say that infinite binary words w and w' are synchronizable, and we write w \equiv_{\text{s}} w', iff there exists \(d, d' \in \mathbb{N}\) such that \(w' \leq \mathbf{0}^d w'\) and \(w' \leq \mathbf{0}^{d'} w'\). It means that we can delay w by \(d\) ticks so that the Is of \(w'\) occur before the Is of \(w\), and reciprocally.

It means that the n-th 1 of w is at a bounded distance from the n-th 1 of w'. E.g., 1(10) and (01) are synchronizable; 11(0) and (00) are not synchronizable; (0110) and (10) are not synchronizable since there are asymptotically too many reads or writes.

In the case of periodic clocks, the notion of synchronizability is computable.

**Proposition 6.** Two infinite periodic binary words w = u(v) and w' = u'(v') are synchronizable, denoted by w \equiv_{\text{p}} w', iff they have the same rate (a.k.a. throughput): \[|v|/|v'| = |v'|/|v|\].

In other words, w \equiv_{\text{p}} w' means w and w' have the same fraction of 1s in (v) and (v'), hence the same asymptotic production rate. It also means the n-th 1 of w is at a bounded distance from the n-th 1 of w'.

**Proof.** From Remark 1, consider \(w_1 = u(v)\) and \(w_2 = u'(v')\) with |u| = |u'| and |v| = |v'|, \(w_1 = u(v) \equiv_{\text{p}} w_2 = u'(v')\) iff there exists \(d, d'\) s.t. \forall w \leq w_2[1..|u| + |v| + d], w' \leq \mathbf{0}^{d'} w'_2[1..|u| + |v| + \text{lcm}(|u|, |v|)] \wedge |w| = |w'| = |v'|. By \(w_1 \equiv_{\text{p}} w_2\) \iff |v'|/|v| = |v'|/|v|, we have \(w_1 \equiv_{\text{p}} w_2\) \iff |v'|/|v| = |v'|/|v|. It is sufficient to cover the prefixes of finite length \(|u| + |v| + \text{lcm}(d, d')\).

If \(|v'|/|v| = 0\) is straightforward. Let us assume that \(|v'|/|v| < |v'|/|v|\) (the case \(|v'|/|v| > |v'|/|v|\) is symmetric). Because of Remark 1, it means \(|v'|/|v| > |v'|/|v|\). Then it entails that (v') and (v) are not synchronizable so as w_1 and w_2. Let us denote a = |v'|/|v| \geq a then v has no 1 more than v'. Thus \(v' \leq \mathbf{0}^{n+1} w\) where \(v' < |v'| - 1\).

**Proposition 5** (on-monotonicity). For any given infinite binary word w, functions \(x \mapsto x \cdot w\) on x and \(x \mapsto w\) on x are monotone. The latter is also injective but not the former.\(^3\)

Using infinite binary words, we can exhibit an interesting set of clocks that we call ultimately periodic clocks or simply periodic clocks. A periodic clock is a clock whose stream is periodic. Periodic clocks are defined as follows:

\[c ::= w \mid c \cdot w, \quad w \in \mathbb{Q}_2\]

In the case of these periodic clocks, Proposition 2 becomes an algorithm, allowing to effectively compute the result of \(c \cdot w\). Let us consider two infinite periodic binary words \(w_1 = u_1(v_1)\) and \(w_2 = u_2(v_2)\) with \(|v_1| = |v_2|\) and \(|v_1| = |v_2|\), this is possible because of Remark 1. Then \(w_3 \equiv_{\text{p}} w_1 \cdot w_2 = u_3(v_3)\) is computed by \(|v_3| = |v_1| + |v_2| = |u_1| + |u_2|\). Indeed, \(v_3 = |v_1| \cdot |v_2| = |v_1| \cdot |v_2| = |v_1| = |v_2|\).

Likewise, periodic clocks are closed for the pointwise extensions of boolean operators \&, \|, \&, and \|.\(^3\)

\(^3\)E.g., (00110(10) = (11100) \cdot (10).
\[ |u|_1 k = \max(0, |w_1| - |w'|_1) \]. (v), \( p = \min(\{ |f| \leq |u| + r \land \gamma' u \leq 0 u' \}) \) and \( q = \min(\{ |f| \leq |u| + \lambda' u \leq 0 u \}) \). (v') are also synchroniz-
\[ \text{able, thus } (v) \leq 0^
\text{in} (v') \text{ and } (v') \leq 0\text{in}(v) \). Then \( w_1 \leq 0^{\text{in} + r\text{in}'} (v)\text{, and } w_2 \leq 0^{\text{in} + r\text{in}'} (v) \). There is an additional delay of \( r|v| \) since each period \( v \) holds at least one \( 1 \).

4. The Programming Language

This section introduces a simple data-flow functional language on infinite data streams. The semantics of this language has a strictly synchronous core, enforced by a so-called clock calculus, a type system to reject non synchronous programs, following [8, 13]. Our main contribution is to extend this core with a relaxed interpretation of synchrony. This is obtained by extending the clock calculus so as to accept the composition of streams whose clocks are “almost equal”. These program can in turn be automatically transformed into conventional synchronous programs by inserting buffer code at proper places.

4.1 A Synchronous Data-Flow Kernel

We introduce a core data-flow language on infinite streams. Its syntax derives from [12]. Expressions (\( e \)) are made of constant streams (\( i \)), variables (\( x \)), pairs (\( e,e \)), local definitions of functions or stream variables (\( e \ where \ x = e \)), applications (\( e(e) \)), initialized delays (\( e \ \text{tby} \ e \)) and the following sampling functions: \( e \ when \ pe \) is the sampled stream of \( e \) on the periodic clock given by the value of \( pe \), and \( merge \) is the combination operator of complementary streams (with opposite periodic clocks) in order to form a longer stream; \( \text{fst} \) and \( \text{snd} \) are the classical access functions. As a syntactic sugar, \( e \ whenot \ pe \) is the sampled stream of \( e \) on the negation of the periodic clock \( pe \).

A program is made of a sequence of declarations of stream functions \( \text{let node \ f \ x = e} \) and periodic clocks (period \( p = pe \)). E.g., \( \text{period half } = \{01\} \) defines the half periodic clock (the alternating bit sequence) and this clock can be used again to build an other one like \( \text{period quarter } = \text{half on half} \). Periodic clocks can be combined with boolean operators. Note that clocks are \textit{static} expressions which can be simplified at compile time into the normal form \( u(v) \) of infinite periodic binary words.

\[
\begin{align*}
\sigma &::= x | \{e, e\} | e \ when \ x = e \ | e(e) | \text{ap}(e, e) | \text{tby}(e, e) | \text{when}(e, e) | \text{merge}(e, e) | \text{fst}(e) | \text{snd}(e) | e \ at \ e.
\end{align*}
\]

\[
\begin{align*}
d &::= \text{let \ node \ f \ x = e | d; d} \\
dp &::= \text{period \ p = pe | dp; dp} \\
pe &::= p \ | pe \ on \ pe | \text{not}(pe) | \text{or}(pe) | \text{and}(pe) | \text{pe} \ \text{in} \ pe.
\end{align*}
\]

We can easily program the downscaler in this language, as shown in Figure 3. The main function consists in composing the various filtering functions. Notation \( o \ at \ (i \ when \ (1000000)) \) is a constraint given by the programmer; it states that the output pixel \( o \) must be produced at some clock \( \alpha \ on \ (1000000) \), thus 6 times slower than the input clock \( \alpha \).

\[
\begin{align*}
\text{let \ period \ c = \{10100100\}} \\
\text{let \ node \ hf \ p = o \ where \ rec \ (...) \text{ and } o = f \ \{p,0,0,3,0,4,0,5,6\} \text{ when } c} \\
\text{let \ node \ main \ i = o \ at \ (i \ when \ (1000000)) \text{ where } rec} \\
\text{ t = hf \ i} \\
\text{ and \(\{11,12,13,14,15,16\} = \text{reorder} \ t \text{ and } o = \text{vf} \ \{11,12,13,14,15,16\}.)}
\end{align*}
\]

Figure 3. Synchronous code using periodic clock

4.2 Synchronous Semantics

The (synchronous) denotational semantics of our core data-flow language is built on classical theory of synchronous languages [12]. Up to syntactic details, this is essentially the core LUSTRE language. Nonetheless, to ease the presentation, we have restricted sampling operations to apply to periodic clocks only (whereas any boolean sequence can be used to sample a stream in existing synchronous languages). Moreover, these periodic clocks are defined globally as constant values. These period expressions can in turn be automatically transformed into plain synchronous code or circuits (i.e., expressions from \( e \)) [25].

This kernel can be statically typed with straightforward typing rules [12]; we will only consider clock types in the following. In the same way, we do not consider causality and initialization problems nor the rejection of recursive stream functions. These classical analyses apply directly to our core language and they are orthogonal to synchrony.

The compilation process takes two steps.

1. A clock calculus computes all constraints satisfied by every clock, as generated by a specific type system. These constraints are resolved through a unification procedure, to infer a periodic clock for each expression in the program. If there is no solution, we prove that some expressions do not have a periodic execution consistent with the rest of the program: the program is not synchronous, and therefore is rejected.

2. If a solution is found, the code generation step transforms the data-flow program into an imperative one (executable, OCaml, etc.) where all processes are synchronously executed according to their actual clock.

4.2.1 Clock Calculus

We propose a type system to generate the clock constraints. The goal of the clock calculus is to produce judgments of the form \( P, H \vdash e : ct \) meaning that “the expression \( e \) has clock type \( ct \) in the environments of periods \( P \) and the environment \( H \).”

Clock types\(^5\) are split into two categories, clock schemes (\( \sigma \)) quantified over a set of clock variables (\( \alpha \)) and unquantified clock types (\( ct \)). A clock may be a functional clock (\( ct \to ct \)), a product (\( ct \times ct \)) or a stream clock (\( ck \)). A stream clock may be a sampled clock (\( ck \ on \ pe \)) or a clock variable (\( \alpha \)).

\[
\begin{align*}
\sigma &::= \forall \alpha_1, ..., \alpha_n. ct \\
ct &::= ct \to ct | ct \times ct | ck \\
ck &::= ck \ on \ pe \ | \alpha. \\
H &::= [x; \sigma_1; ..., \sigma_n; \sigma_0] \\
P &::= [p; pe; ..., p; pe]\n\end{align*}
\]

The distinction between clock types (\( ct \)) and stream clock types (\( ck \)) should not surprise the reader. Indeed, whereas Kahn networks do not have clock types [18], there is a clear distinction between a channel (which receives some clock type \( ck \)), a stream function (which receives some functional clock type \( ct \to ct' \)) and a pair expression (which receives some clock type \( ct \times ct' \) meaning that the two expressions do not necessarily have synchronized values).

Clocks may be instantiated and generalized. This is a key feature, to achieve modularity of the analysis. E.g., the horizontal fil-

\[\text{ter of the downscaler has clock scheme } \forall \alpha. \alpha \to \alpha \ on \ (10100100) \; ;
\]

This means that, if the input has any clock \( \alpha \), then the output has some clock \( \alpha \ on \ (10100100) \). This clock type can in turn be instantiated in several ways, replacing \( \alpha \) by more precise stream clock type (e.g., \( \text{some sampled clock } \alpha' \ on \ (01) \)).

\(^5\) We shall sometimes say \( \text{clock} \) instead of \( \text{clock type} \) when clear from context.
The rules for instantiating and generalizing a clock type are given below. $FV(ct)$ denotes the set of free clock variables in $ct$.

\[
e_{1}[\hat{\{k/A\}}] \leq \forall \bar{a}.e_{1} \\
\text{fgen}(ct) = \forall \alpha_{1}, ..., \alpha_{m}.ct \text{ where } \alpha_{1}, ..., \alpha_{m} = FV(ct)
\]

It states that a clock scheme can be instantiated by replacing variables with clock expressions. $\text{fgen}(ct)$ returns a fully generalized clock type where every variable in $ct$ is quantified universally.

When defining periods, we must take care that identifiers are already defined. If $P$ is a period environment (i.e., a function from period names to periods), we shall simply write $P\vdash pe$ when every free name appearing in $pe$ is defined in $P$.

The clocking rules defining the predicate $P,H \vdash e : ct$ are now given in Figure 4 and are discussed below.

\[
\begin{align*}
\text{(IM)} & \quad P,H \vdash i : ck \\
\text{(INST)} & \quad ct \leq H(x) \quad P,H \vdash x : ct \\
\text{(OP)} & \quad P,H \vdash e_{1} : ck \quad P,H \vdash e_{2} : ck \\
\text{(FBY)} & \quad P,H \vdash op(e_{1}, e_{2}) : ck \\
\text{(FBY)} & \quad P,H \vdash e_{1} : ck \quad P,H \vdash e_{2} : ck \\
\text{(WHEN)} & \quad P,H \vdash e : ck \quad P \vdash pe \\
\text{(MERGE)} & \quad P \vdash pe \quad H \vdash e_{1} : ck \text{ on } pe \quad P,H \vdash e_{2} : ck \text{ on } not \text{ pe} \\
\text{(APP)} & \quad P,H \vdash e_{1} : ct_{2} \rightarrow ct_{1} \quad P,H \vdash e_{2} : ct_{2} \\
\text{(PAIR)} & \quad P,H \vdash e_{1} : ct_{1} \quad P,H \vdash e_{2} : ct_{2} \\
\text((FST)} & \quad P,H \vdash e : ct_{2} \times ct_{2} \quad P,H \vdash \text{fst } e : ct_{2} \\
\text{(SND)} & \quad P,H \vdash e : ct_{1} \times ct_{2} \\
\text{(CTR)} & \quad P,H \vdash e_{1} : ck \quad P,H \vdash e_{2} : ct_{2} \\
\text{(NODE)} & \quad P,H \vdash e_{2} \text{ at } e_{1} : ck \\
\text{(PERIOD)} & \quad P \vdash pe \\
\text{(DEFH)} & \quad H \vdash d_{1} : H_{1} \quad H_{1}, H_{2} \vdash d_{2} : H_{2} \\
\text{(DEEP)} & \quad P \vdash d_{1} : P_{1} \quad P \vdash d_{2} : P_{2} \\
\end{align*}
\]

\textbf{Figure 4.} The core clock calculus

- A constant stream may have any clock $ck$ (rule (IM)).
- The clock of an identifier can be instantiated (rule (INST)).
- The inputs of imported primitives must all be on the same clock (rule (OP)).
- Rule (FBY) states that the clock of $e_{1}$ by $e_{2}$ is the one of $e_{1}$ and $e_{2}$ (they must be identical).
- Rule (WHEN) states that the clock of $e$ when $pe$ is a sub-clock of the clock of $e$ and we write it $ck$ on $pe$. In doing so, we must check that $pe$ is a valid periodic clock.
- Rule (MERGE) states an expression $merge pe e_{1} e_{2}$ is well clocked and on clock $ck$ if $e_{1}$ is on clock $ck$ on $pe$ and $e_{2}$ is on the complementary clock $ck$ on not $pe$.
- Rule (APP) is the classical typing rule of ML type systems.
- Rule (WHERE) is the rule for recursive definitions.
- Rules (PAIR), (FST) and (SND) are the rules for pairs.
- Rule (CTR) for the syntax $e_{1} at e_{2}$ states that the clock associated to $e_{1}$ is imposed by the clock of $e_{2}$; it is the type constraint for clocks.
- Node declarations (rule (NODE)) are clocked as regular function definitions. We write $H, x : ct_{1}$ as the clock environment $H$ extended with the association $x : ct_{1}$. Because node definitions only apply at top-level (and cannot be nested), we can generalize every variable appearing in the clock type.\(^{5}\)
- Rules (PERIOD), (DEFH) and (DEEP) check that period and stream variables are well formed, i.e., names in period and stream expressions are first defined before being used.

4.2.2 Structural Clock Unification

In synchronous data-flow languages such as LUSTRE or LUCID SYNCHRONIE, clocks can be made of arbitrarily complex boolean expressions. In practice, the compiler makes no hypothesis on the condition $c$ in the clock type $(ck on c)$. This expressiveness is an essential feature of synchronous languages but forces the compiler to use a syntactical criteria during the unification process: two clock types $(ck_{1} on c_{1})$ and $(ck_{2} on c_{2})$ can be unified if $ck_{1}$ and $ck_{2}$ can be unified and if $c_{1}$ and $c_{2}$ are syntactically equal.

This approach can also be applied in the case of periodic clocks. Two clock types $(ck on w_{1})$ and $(ck on w_{2})$ can be unified if $ck_{1}$ and $ck_{2}$ can be unified and if $w_{1} = w_{2}$ (for the equality between infinite binary words). As a result, this structural clock unification is unable to compare $(ck on (01))$ on $01$ and $\alpha on (0001)$ though two stream on these clocks are present and absent at the very same instants. A more clever unification mechanism will be the purpose of section 4.3.4.

4.2.3 Semantics over Clocked Streams

We provide our language with a data-flow semantics over finite and infinite sequences following Kahn formulation [18]. Nonetheless, we restrict the Kahn semantics by making the absence of a value explicit. The set of instantaneous values is enriched with a special value $\bot$ representing the absence of a value.

We need a few preliminary notations. If $T$ is a set, $T^{\omega}$ denotes the set of finite or infinite sequences of elements over the set $T (T^{\omega} = T^{+} + T^{0})$. The empty sequence is noted $\varepsilon$ and $x_{\varepsilon}x$ denotes the sequence whose head is $x$ and tail is $\varepsilon$. Let $\leq$ be the prefix order over sequences, i.e., $x \leq y$ if $x$ is a prefix of $y$. The ordered set $\mathbb{D} = (T^{\omega}, \preceq)$ is a complete partial order (CPO). If $D_{1}$ and $D_{2}$ are CPOs, then $D_{1} \times D_{2}$ is a CPO with the coordinate-wise order. $[D_{1} \to D_{2}]$ as the set of continuous functions from $D_{1}$ to $D_{2}$ is also a CPO by taking the pointwise order. If $f$ is a continuous mapping from $D_{1}$ to $D_{2}$, we shall write $\text{fix}(f) = \lim_{h \to \infty} f^{h}(\varepsilon)$ for the smallest fix point of $f$ (Kleene theorem). We define the set

\[^{5}\text{This is slightly simpler than the classical generalization rule of ML which must restrict the generalization to variables which do not appear free in the environment.}\]
ClockedStream\( (T) \) of clocked sequences as the set of finite and infinite sequences of elements over the set \( T = T \cup \{ \bot \} \).

\[
\begin{align*}
\text{ClockedStream}(T) & = T \cup \{ \bot \} \\
\text{ClockedStream}(T) & = (T_\bot)^* 
\end{align*}
\]

A clocked sequence is made of present or absent values. We define the clock of a sequence \( s \) as a boolean sequence (without absent values) indicating when a value is present. For this purpose, we define the function \( \text{clock} \) from clocked sequences to boolean sequences:

\[
\begin{align*}
\text{clock}(\varepsilon) & = \varepsilon \\
\text{clock}(\bot.s) & = 0.\text{clock}(s) \\
\text{clock}(s.s) & = 1.\text{clock}(s)
\end{align*}
\]

We shall use the letter \( v \) for present values. Thus, \( v.s \) denotes a stream whose first element is present and whose rest is \( s \) whereas \( \bot.s \) denotes a stream whose first element is absent. The interpretation of basic primitives of the core language over clocked sequences is given in figure 5. We use the mark \# to distinguish the syntactic construct (e.g., \( \text{fby} \)) from its interpretation as a stream transformer:

\[
\begin{align*}
\text{const}^\# i 1.c & = i.\text{const}^\# i c \\
\text{const}^\# 0 1.c & = \bot.\text{const}^\# 0 c \\
op^\# (s_1, s_2) & = \varepsilon \text{ if } s_1 = \varepsilon \text{ or } s_2 = \varepsilon \\
op^\# (\bot.s_1, \bot.s_2) & = \bot.\text{op}^\# (s_1, s_2) \\
op^\# (v_1.s_1, v_2.s_2) & = (v_1 \text{ op} v_2).\text{op}^\# (s_1, s_2) \\
fby^\# (\varepsilon, s) & = \varepsilon \\
fby^\# (\bot.s_1, \bot.s_2) & = \bot.\text{fby}^\# (s_1, s_2) \\
fby^\# (v_1.s_1, v_2.s_2) & = v_1.\text{fby}^\# (s_1, s_2) \\
fby^\# (\bot, v_1.s_1, v_2.s_2) & = \bot.\text{fby}^\# (v_1.s_1, v_2.s_2) \\
fby^\# (v, \bot.s_1, v_2.s_2) & = v.\text{fby}^\# (v_1.s_1, v_2.s_2) \\
\text{when}^\# (\varepsilon, c) & = \varepsilon \\
\text{when}^\# (\varepsilon.s, c) & = \bot.\text{when}^\# (c, s) \\
\text{when}^\# (v.s, 1.c) & = v.\text{when}^\# (s, c) \\
\text{when}^\# (v.s, 0.c) & = \bot.\text{when}^\# (s, c) \\
\text{merge}^\# (c, s_1, s_2) & = \varepsilon \text{ if } s_1 = \varepsilon \text{ or } s_2 = \varepsilon \\
\text{merge}^\# (c, v.s_1, \bot.s_2) & = v.\text{merge}^\# (c, s_1, s_2) \\
\text{merge}^\# (c, \bot.s_1, v.s_2) & = v.\text{merge}^\# (c, s_1, s_2) \\
\text{not}^\# 1.c & = 0.\text{not}^\# c \\
\text{not}^\# 0.c & = 1.\text{not}^\# c \\
\text{on}^\# (1.c_1, 1.c_2) & = 1.\text{on}^\# (c_1, c_2) \\
\text{on}^\# (1.c_1, 0.c_2) & = 0.\text{on}^\# (c_1, c_2) \\
\text{on}^\# (0.c_1, 1.c_2) & = 0.\text{on}^\# (c_1, c_2)
\end{align*}
\]

Figure 5. Semantics for the core primitives

- The \text{const} primitive produces a constant stream from an immediate value. This primitive is polymorphic since it may produce a value (or not) according to the environment. For this reason, we add an extra argument giving its clock. Thus, \text{const}^\# i c denotes a constant stream with stream clock \( c \) (\text{clock}(\text{const}^\# i c) = c).

- For a binary operator, the two operands must be synchronous (together present or together absent) and the purpose of the clock calculus is to ensure it statically (otherwise, some buffering is necessary).

- \text{fby} is the unitary delay: it “conses” the head of its first argument to its second one. The arguments and result of \text{fby} must be on the same clock. \text{fby} corresponds to a two-state machine: while both arguments are absent, it emits nothing and stays in its initial state (\text{fby}^\#). When both are present, it emits its first argument and enters the new state (\text{fby}^1\#) storing the previous value of its second argument. In this state, it emits a value every time its two arguments are present.

- The clocking operator expects two arguments on the same clock. The clock of the result depends on the boolean condition (c).

- The definition of \text{merge} states that one branch must be present when the other is absent.

- Note that \text{not}^\# and \text{on}^\# operate on boolean sequences only. The other boolean operations on clocks, e.g. or and \&, follow the same principle.

It is easy to check that all these functions are continuous on clocked sequences. Semantics is given to expressions which have passed the clock calculus (\text{e} judgments). We define the interpretation of clock types as the following:

\[
\begin{align*}
[ct_1 \rightarrow ct_2] & = [[ct_1]_p \rightarrow [[ct_2]_p]_p] \\
[ct_1 \times ct_2] & = [[ct_1]_p \times [[ct_2]_p]_p] \\
\{v \in \mathbb{V} ; \ldots ; v_n \in \mathbb{V} \} & = \langle v_1 \ldots , v_n \rangle \\
[\text{clock}(s)] & = \text{clock}(s) \leq P(c)
\end{align*}
\]

In order to take away causality problems (which are treated by some dedicated analysis in synchronous languages), \([ck]_p\) contains all the streams whose clock is a prefix of the value of \text{ck} (and in particular the empty sequence \(\varepsilon\)). This way, an equation \( x = x + 1 \) which is well clocked (since \( P.H \cdot x : ck \rightarrow x + 1 : ck \)) but not causal (its smallest solution is \(\varepsilon\)) can receive a synchronous semantics.

For any environment \( P \), clock environment \( H \) and any assignment \( \rho \) (which maps variable names to values) such that \( \rho(\varepsilon) \in [H(x)]_p \), the meaning of an expression is given by \( [P,H \cdot e : ct]_p \) such that \( [P,H \cdot e : ct]_p \in [ct]_p \). The denotational semantics of the language is defined structurally in Figure 6.

4.2.4 Example

Let us illustrate these definitions on the downscaler in Figure 3.

1. Suppose that the input \( i \) has some clock type \( \alpha_i \).

2. The horizontal filter has the following signature, corresponding to the effective synchronous implementation of the process: \( \alpha_2 \rightarrow \alpha_2 \) on \( (10101\,0110) \).

3. Between the horizontal filter and the vertical filter, the reorder process stores the 5 previous lines in a sliding window of size 5, but has no impact on the clock besides delaying the output until it receives 5 full lines, i.e., \( 5 \times 720 = 3600 \) cycles. We shall give to the reorder process the clock signature \( \alpha_3 \rightarrow \alpha_3 \) on \( 0^{660}(1) \).

4. The vertical filter produces 4 pixels from 9 pixels repeatedly across the 720 pixels of a stripe (6 lines). Its signature (matching the process’s synchronous implementation) is:

\[
\alpha_4 \rightarrow \alpha_4 \text{ on } (1^{720}0/720^{720}0/720^{720}0/720^{720}0/720^{720})
\]

To simplify the presentation, we will assume in manual computations that the unit of computation of the vertical filter is a line and not a pixel, hence replace 720 by 1 in the previous signature, yielding: \( \alpha_4 \rightarrow \alpha_4 \) on \( (10101\,0110) \).

5. Finally, the designer has required that if the global input \( i \) is on clock \( \alpha_4 \), then the clock of the output \( o \) should be \( \alpha_1 \) on \( (100000) \) — the 6 times subsampled input clock — tolerating an additional delay that must automatically be deduced from the clock calculus.
The composition of all 5 processes yield the type constraints \( \alpha_1 = \alpha_2, \alpha_3 = \alpha_2, \alpha_4 = \alpha_3 \) on \( 0^{1001001}(1) \). After replacing variables by their definitions, we get for the output \( o \) the following clock type:
\[
((\alpha_1 \text{ on } 0^{1001001}(1)) \text{ on } 0^{1001001}(1)) = \alpha_1 \text{ on } 0^{1001001}(100000000000000000000000001).
\]

Yet, the result is not equal to the clock constraint stating that \( o \) should have clock type \( \alpha_1 \) on \( 0^{1000000} \). The downscaler is thus rejected in a conventional synchronous calculus. This is the reason why we introduce the relaxed notion of synchronizability.

### 4.3 Relaxed Synchronous Semantics

The downscaler example highlights a fundamental problem with the embedding of video streaming applications in a synchronous programming model. The designer often has good reasons to apply a synchronous operator (e.g., the addition) on two channels with different clocks, or to compose two synchronous processes whose signatures do not match, or to impose a particular clock which does not match any solution of the constraints equations. Indeed, in many cases, the conflicting clocks may be “almost identical”, i.e., they have the same asymptotic production rate. This advocates for a more relaxed interpretation of synchronism. Our main contribution is a clock calculus to accept the composition of clocks which are “almost identical”, as defined by the structural extension of the synchronizability relation on infinite binary words to stream clocks:

**Definition 2** (synchronizable clocks). We say that two stream clocks \( c_k \) on \( w \) and \( c_k \) on \( w' \) are synchronizable, and we write \( c_k \) on \( w \equiv c_k \) on \( w' \), if and only if \( w \equiv w' \).

Notice this definition does not directly extend to stream clocks with different variables.

### 4.3.1 Buffer Processes

When two processes communicate with synchronous clocks, and when causality is preserved (i.e., writes precede or coincide with reads), one may effectively generate synchronous code for storing (the bounded number of) pending writes.

Consider two infinite binary words \( w \) and \( w' \) with \( w \equiv w' \). A buffer \( buffer_{w,w'} \) is a process with the clock type \( buffer_{w,w'} : \forall \beta \beta w \rightarrow \beta \beta w' \) and with the data-flow semantics of an unbounded lossless FIFO channel [18]. The existence of such an (a priori unbounded) buffer is guaranteed by the causality of the communication (writes occur at clock \( w \) that precede clock \( w' \)). We are only interested in buffers of finite size (a.k.a. bounded buffers), where the size of a buffer is the maximal number of pending writes it can accomodate while preserving the semantics of an unbounded lossless FIFO channel.

**Proposition 7.** Consider two processes \( f : \beta \rightarrow \alpha \) on \( w \) and \( f' : \alpha \rightarrow \alpha' \) on \( w' \), with \( w \equiv w' \) and \( w \not\equiv w' \). There exists a buffer \( buffer_{w,w'} : \forall \beta \beta w \rightarrow \beta \beta w' \) such that \( f' / buffer_{w,w'} f \) is a \((0-)\)-synchronous composition (with the unification \( \alpha = \beta \)).

**Proof.** A buffer of size \( n \) can be implemented with \( n \) data registers \( x_i \) and \( 2n + 1 \) clocks \( (w_1)_{1 \leq i \leq n} \) and \((r_i)_{0 \leq i \leq n} \). Pending writes are stored in data registers: \( w_i[j] = 1 \) means that there is a pending write stored in \( x_i \) at cycle \( j \). Clocks \( r_i \) determine the instants when the process associated with \( w_i \) reads the data in \( x_i \) at \( r_i[j] = 1 \) means that the data in register \( x_i \) is read at cycle \( j \). For a sequence of pushes and pops imposed by clocks \( w_i \) and \( w' \), the following case distinction simulates a FIFO on the \( x \) registers statically controlled through clocks \( w_i \) and \( r_i \):

**NOP:** \( w_i[j] = 0 \) and \( w'[j] = 0 \), No operation affects the buffer, i.e., \( r_i[j] = 0, w_i[j] = w_i[j - 1]; \) registers \( x \) are left unchanged.

**PUSH:** \( w_i[j] = 1 \) and \( w'[j] = 0 \). Some data is pushed into the buffer and stored in register \( x_i \), all the data in the buffer being pushed from \( x_i \) into \( x_i + 1 \). Thus \( x_i = x_i + 1 \) and \( s_i = \) input, \( \forall i, 2 \leq w_i[j] = w_i[j - 1], w_i[j] = 1 \) and \( r_i[j] = 0 \).

**POP:** \( w_i[j] = 0 \) and \( w'[j] = 0 \) and \( w'[j] = 1 \). Let \( p = \max(\{0 \cup \{1 \leq i \leq n \mid w_i[j] = 1 \}) \). If \( p \) is zero, then no register stores any data at cycle \( j \); input data must be bypassed directly to the output, crossing the wire clocked by \( r_0 \), setting \( r_i[j] = 0 \) for \( i > 0 \) and \( r_i[j] = 1, w_i[j] = w_i[j - 1] \). Conversely, if \( p > 0, \forall i \neq p, r_i[j] = 0, r_i[j] = 1, x_i[j] = x_i[j - 1] \), and \( w_i[j] = 0 \). Registers \( x_i \) are left unchanged (notice this is not symmetric to the PUSH operation).

**POP; PUSH:** \( w_i[j] = 1 \) and \( w'[j] = 1 \). This case boils down to the implementation of a \( P \) followed by a \( P \), as defined in the previous two cases.

Assuming \( w \) and \( w' \) are periodic and have been written \( w = u(v) \) and \( w' = u'(v') \), under the lines of Remark 1, it is sufficient to conduct the previous simulation for \(|u| + |v| \) cycles to compute periodic clocks \( w_i \) and \( r_i \). This leads to an implementation in a plain \((0-)\)-synchronous language; yet this implementation is impractical because each clock \( w_i \) or \( r_i \) has a worst case quadratic size in the maximum of the periods of \( w \) and \( w' \) (from the application of remark 1), yielding cubic control space, memory usage and code size. This motivates the search for an alternative buffer implementation decoupling the memory management for the FIFO from the
combinatorial control space; such an implementation is proposed in Section 5.2.

4.3.2 Relaxed Clock Calculus

Let us now modify the clock calculus in two ways:

1. a subtyping [22] rule (SUB) is added to the clock calculus to permit the automatic insertion of a finite buffer in order to synchronize clocks;

2. rule (CTR) is modified into a subtyping rule to allow automatic insertion (and calculation) of a bounded delay.

**The Subtyping Rule**

**Definition 3.** The relation $<$ is defined by

$$ w <: w' \iff w \preceq w' \wedge w \preceq w'. $$

This is a partial order, and its restriction to equivalence classes for the synchronizability relation ($\equiv$) forms a complete lattice.

We structurally extend this definition to stream clocks $ck$ on $w$ and $ck$ on $w'$ where $w <: w'$.

Relation $<$: defines a subtyping rule (SUB) on stream clocks types:

$$ (\text{SUB}) \begin{align*} P,H \vdash e : ck \text{ on } w \quad &w <: w' \quad \Rightarrow \quad P,H \vdash e : ck \text{ on } w' \\ P,H \vdash e : ck \text{ on } w' \end{align*} $$

This is a standard subsumption rule, and all classical results on subtyping apply [22].

The clock calculus defined in the previous section rejects expressions such as $x + y$ when the clocks of $x$ and $y$ cannot be unified. With rule (SUB), we can relax this calculus to allow an expression $e$ with clock $ck$ to be used “as if it had” clock $ck'$ as soon as $ck$ and $ck'$ are synchronizable and causality is preserved.

E.g., the following program is rejected in the $(\neg \neg)$-synchronous calculus, assuming $x$ has some clock $\alpha$, $\alpha$ on $(01)$ cannot be unified with $\alpha$ on $(10)$.

$$ \begin{align*} \text{let } f(x) &= y \text{ where} \\ y &= (x \text{ when } (01)) + (x \text{ when } (10)) \end{align*} $$

Let $e_1$ denote expression $(x \text{ when } (01))$ and $e_2$ denote expression $(x \text{ when } (10))$, and let us generate the type constraints for each construct in the program:

1. (NODE): suppose that the signature of $f$ is of form $f : \alpha \rightarrow \alpha'$;

2. (*): the addition expects two arguments with the same clocks;

3. (WHEN): we get $ck_1 = \alpha$ on $(01)$ for the clock of $e_1$ and $ck_2 = \alpha$ on $(10)$ for the clock of $e_2$;

4. (SUB): because $(01)$ and $(10)$ are synchronizable, the two clocks $ck_1 = \alpha$ on $(01)$ and $ck_2 = \alpha$ on $(10)$ can be resynchronized into $\alpha$ on $(01)$, since $(01) <: (10)$ and $(10) <: (01)$.

The final signature is $f : \forall \alpha. \alpha \rightarrow \alpha$ on $(01)$.

Considering the downscalet example, this subtyping rule (alone) does not solve the clock conflict: the imposed clock first needs to be delayed to avoid starvation of the output process. This is the purpose of the following rule.

**The Clock Constraint Rule**

The designer may impose the clock of certain expressions. Rule (CTR) is relaxed into the following subtyping rule:

$$ (\text{CTR}) \begin{align*} P,H \vdash e_1 : ck \text{ on } w_1 &\quad P,H \vdash e_2 : ck \text{ on } w_2 \quad \Rightarrow \quad w_1 <: 0^d w_2 \\ P,H \vdash e_1 \text{ at } e_2 : ck \text{ on } 0^d w_2 \end{align*} $$

Consider the previous example with the additional constraint that the output must have clock $(1001)$.

$$ \begin{align*} \text{let } f(x) &= y \text{ at } (x \text{ when } (1001)) \text{ where} \\ y &= (x \text{ when } (01)) + (x \text{ when } (10)) \end{align*} $$

We previously computed that $(x \text{ when } (01)) + (x \text{ when } (10))$ has signature $\alpha \rightarrow \alpha$ on $(01)$, and $(01)$ does not unify with $(1001)$.

Rule (CTR) yields

$$ P,H \vdash y : a \text{ on } (01), x \text{ when } (1001) : a \text{ on } (1001) \quad (01) <: (00101) $$

Finally, $f : \forall \alpha. \alpha \rightarrow \alpha$ on $(01001)$. Indeed, one cycle delay is the minimum to allow synchronization with the imposed output clock.

**Relaxed Clock Calculus Rules**

The predicate $P,H \vdash e : ct$ states that an expression $e$ has clock $ct$ in the period environment $P$ and the clock environment $H$, under the use of some synchronization mechanism. Its definition extends the one of $P,H \vdash e : ct$ with the new rules in Figure 7. The axiom and all other rules are identical to the ones in Figure 4, using $\vdash_e$ judgments instead of $\vdash$.

$$ (\text{SUB}) \begin{align*} P,H \vdash e : ck \text{ on } w_1 &\quad w_1 <: w_2 \\ P,H \vdash e : ck \text{ on } w_2 \end{align*} $$

$$ (\text{CTR}) \begin{align*} P,H \vdash e_1 : ck \text{ on } w_1 &\quad P,H \vdash e_2 : ck \text{ on } w_2 \quad \Rightarrow \quad w_1 <: 0^d w_2 \\ P,H \vdash e_1 \text{ at } e_2 : ck \text{ on } 0^d w_2 \end{align*} $$

Figure 7. The relaxed clock calculus

Thus, starting from a standard clock calculus whose purpose is to reject non-synchronous program, we extend it with subtyping rules expressing that a stream produced on some clock $ck_1$ can be read on the clock $ck_2$ as soon as $ck_1$ can be synchronized into $ck_2$, using some buffering mechanism. By presenting the system in two steps, the additional expressiveness with respect to classical synchrony is made more precise.

**Relaxed Synchrony and the $\mathbf{by}$ Operator**

Notice $\mathbf{by}$ is considered a length preserving function in data-flow networks, hence its clock scheme $\forall \alpha. \alpha \times \alpha \rightarrow \alpha$ in the $\neg \neg$-synchronous case, and despite it only needs its first argument at the very first instant. In the relaxed case, we could have chosen one of the following clock signatures: $\forall \alpha. \alpha$ on $(01) \times \alpha \rightarrow \alpha$, $\forall \alpha. \alpha$ on $(1001) \times \alpha$ on $(01) \rightarrow \alpha$, or $\forall \alpha. \alpha \times \alpha$ on $(01)$ $\rightarrow \alpha$. The first two signatures require the first argument to be present at the very first instant, which is overly restrictive in practice. The third signature is fully acceptable, with the observation that the original length-preserving signature can be reconstructed by applying the subtyping rule $\alpha$ on $(1) <: \alpha$ on $(01)$. This highlights the fact that the $\mathbf{by}$ operator is a one-place buffer.

4.3.3 Construction of the System of Clock Constraints

The system of clock constraints is build from the systematic application of the core rules in Figure 4 and the relaxed calculus rules in Figure 7. All rules are syntax directed except (SUB) whose application is implicit at each (function or operator) composition.

Rule (CTR) is a special case: the clock constraint is built by computing a possible value for the delay $d$. This computation is syntax directed, and we always choose to minimize delay insertion: $\text{delay}(w,w') = \min \{ 1 \mid w <: 0^d w' \}$. When $w <: w'$, no delay is necessary. Note that in general, $\text{delay}(w,w') \neq \text{delay}(w',w)$.

**Proposition 8.** The delay to synchronize an infinite periodic binary word $w$ with an imposed infinite periodic binary word $w'$ can be automatically computed by the formula

$$ \text{delay}(w,w') = \max (\{ w \mid p \} - \{ w' \mid p \}, 0). $$
On periodic words, this delay is effectively computable thanks to Remark 1.

**Proof.** Indeed, let \( d = \max\{\max_p(|[w]|_p - |[w']|_p), 0\} \) and \( v = 0^d w \) we have \( w \leq v \) since for all \( p \), \( |[v]|_p = d + |[w]|_p \). Moreover, \( d \) is minimal: suppose there exists \( p \) such that \( d < |[w]|_p - |[w']|_p \), then \( v' = 0^d w' \) satisfies \( |[v']|_p = d - 1 + |[w]|_p < |[w]|_p \). Thus, \( w \not\leq v' \). \( \square \)

For the simplified downscaler, the minimal delay to desynchronize the vertical filter with the output process is 9603, since 9603 (clock cycles) is the minimal value of \( d \) such that 
\[ 9603(0000100000000000001000000100) < 9603(100000000) \]
For the real downscaler (with fully developed vertical filter signature), we automatically computed that the minimal delay was 12000 to permit communication with the SD output.

### 4.3.4 Unification

We need a better unification procedure on clock types than the structural one (see Section 4.2.2), types to obtain an effective resolution algorithm for this system of constraints. In our case, a syntactic unification would unnecessarily reject many synchronous programs with periodic clocks. We propose a semi-interpreted unification that takes into account the semantics of periodic clocks. More precisely, the unification of two clock types \( ct \) and \( ct' \) can be purely structural on functional and pair types, where no simplification on periodic clocks can be applied, but it has to be aware of the properties of the sampling operator \( \text{on} \) when unifying stream clock types of the form \( c_k \text{on} w \) and \( c_k' \text{on} w' \). Two cases must be considered.

First of all, unifying \( \alpha \text{on} w \) and \( \alpha \text{on} w' \) returns true if and only if \( w = w' \).

In the most general case, assume \( \alpha \) and \( \alpha' \) are clock variables (clocks can be normalised, thanks to the associativity of \( \text{on} \)). Equation \( \alpha \text{on} w = \alpha' \text{on} w' \) always has an infinite number of solutions; these solutions generate an infinite number of different infinite binary words. Intuitively, a periodic sampling of \( w \) consists of the insertion of \( 0s \) in \( w \), in a periodic manner. If \( w \leq w' \), it is always possible to delay the \( p-1 \) th 1 in \( w \) (resp. \( w' \)) until the \( p-1 \) th 1 in \( w' \) (resp. \( w \)) through the insertion of \( 0s \) in \( \alpha \) (resp. \( \alpha' \)). Let us define the subsampling relation \( \leq_{\text{SS}} \), such that

\[
\alpha \leq_{\text{SS}} \alpha' \iff \exists \alpha, \alpha = \alpha a \rightarrow d
\]

Note that if \( \alpha \leq_{\text{SS}} \alpha' \) then \( \alpha' \leq_{\text{SS}} \alpha \), but the converse is not true: \( (01) \leq_{\text{SS}} (001) \) and there is no solution \( \alpha \) such that \( (001) = \alpha (01) \).

**Proposition 9.** Relation \( \leq_{\text{SS}} \) is a partial order.

**Proof.** \( \leq_{\text{SS}} \) is trivially reflexive and transitive. Antisymmetry holds because \( \leq \) is a partial order, and \( \alpha \leq_{\text{SS}} \alpha' \) implies \( \alpha' \leq_{\text{SS}} \alpha \).

In a typical unification scheme, one would like to replace the above type equation by “the most general clock type satisfying the equation”. We will see that there is indeed a most general word \( m \) such that all common subsamples of \( w \) and \( w' \) are subsamples of \( m \) (\( \leq_{\text{SS}} \) is an upper semi-lattice), yet the expression of \( m = \nu \text{on} w = \nu' \text{on} w' \) does not lead to a unique choice for \( m \) and for the maximal unifiers \( \nu \) and \( \nu' \). In fact, there can be an infinite set of such words.

In a strictly synchronous setting, we need to fall back to an incomplete unification scheme (some synchronous programs with periodic clocks will be rejected), choosing one of these solutions. If \( (\nu, \nu') \) is the chosen solution, the unification of \( \alpha \text{on} w \) and \( \alpha' \text{on} w' \) yields a unique clock type \( \alpha \text{on} w = \alpha' \text{on} w' \) and every occurrence of \( \alpha \) (resp. \( \alpha' \)) is replaced by \( \alpha \text{on} \nu \) (resp. \( \alpha \text{on} \nu' \)).

Yet in our relaxed synchronous setting, the most general unifier has an interesting property:

**Proposition 10 (synchronizable unifiers).** Consider \( m, w, w', (v_1, v_1') \) and \( (v_2, v_2') \) such that \( m = v_1 \text{on} w = v_1' \text{on} w' = v_2 \text{on} w = v_2' \text{on} w' \); we have \( v_1 \cap v_2 \subseteq v_1' \cap v_2' \).

This directly derives from Proposition 2.

We can make an arbitrary choice for \((v,v')\) among maximal unifiers, and select one that is easy to compute. Formally, we define the earliest substitutions \( \nu' \) and \( \nu'' \) through the following recurrent equations:

\[
\nu'(0^d 1, 0^d 1, w') = 1^d 0^d 1. \nu'(w, w')
\]

\[
\nu''(0^d 1, 0^d 1, w') = 1^d 0^d 1. \nu''(w, w')
\]

Let \( M(w, w') \) denote the unifier

\[
M(w, w') = \nu'(w, w') \text{on} w = \nu''(w, w') \text{on} w'.
\]

The computation of \( \nu' \) and \( \nu'' \) terminates on periodic words because there are a finite number of configurations (bounded by the product of the period lengths of \( w \) and \( w' \)).

\( \tilde{\text{E.g.}}, \ a \text{on} (1000) = a' \text{on} (0101) : 
\]

\[
\begin{array}{cccccccc}
\text{w} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \ldots \\
\text{w'} & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & \ldots \\
\nu'(w, w') & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
\nu''(w, w') & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & \ldots \\
\end{array}
\]

**Proposition 11.** For all \( w, w', p \),

\[
[M(w, w')]_{p+1} = [M(w, w')]_p + \max\{|[w]|_p - |[w']|_p, |[w']|_p - |[w]|_p\}.
\]

**Proof.** An inductive proof derives naturally from the previous algorithm. In particular, observe that between two consecutive 1s in \( M(w, w') \), the associated subword of either \( v \) or \( v' \) is a sequence of 1s, hence either \( [M(w, w')]_{p+1} - [M(w, w')]_p = |[w]|_p - |[w']|_p \) or \([M(w, w')]_{p+1} - [M(w, w')]_p = |[w']|_p - |[w]|_p\).

In addition, \( M(w, w') \) is the maximum common subsample of \( w \) and \( w' \) and has several interesting properties:

**Theorem 1.** (structure of subsamples). The subsampling relation \( \leq_{\text{SS}} \) forms an upper semi-lattice on infinite binary words, the supremum of a pair of words \( w, w' \) being \( M(w, w') \).

**Proof.** We proceed by induction on the position of the \( p-1 \) th 1. Consider a infinite binary word \( m \) as a word \( w = \nu \text{on} w' \). By construction of \( m \), \( |m|_1 = \max(|w|_1, |w'|_1) \), hence \( |m|_1 \leq |w'|_1 \). Assume all common subsamples of \( w \) and \( w' \) form a complete lower semi-lattice structure for \( \leq_{\text{SS}} \), \( M(w, w') \) being the bottom element.

In addition, \( M(w, w') \) is the maximum common subsample of \( w \) and \( w' \) and has several interesting properties:

**Proposition 10 (synchronizable unifiers).** Consider \( m, w, w', (v_1, v_1') \) and \( (v_2, v_2') \) such that \( m = v_1 \text{on} w = v_1' \text{on} w' = v_2 \text{on} w = v_2' \text{on} w' \); we have \( v_1 \cap v_2 \subseteq v_1' \cap v_2' \).

We may now define a resolution procedure through a set of constraint-simplification rules.

The clock system given is turned into an algorithm by introducing a subtyping rule at every application point and by solving a set of constraints of the form \( c_k \prec c_k' \). The program is well clocked if the set of constraints is satisfiable.
DEFINITION 4 (constraints and satisfiability). A system $S$ of clock constraints is a collection of inequations between clock types:

$$S ::= \{ck_1 < ck'_1, \ldots, ck_n < ck'_n\}$$

We write $S + \{ck_1 < ck'_1\}$ for the extension of a system $S$ with the inequation $\{ck_1 < ck'_1\}$. We write $S \oplus \{ck_1 < ck'_1\}$ for $S + \{ck_1 < ck'_1\}$ such that $S$ does not contain a directed chain of inequations from any free variable in $ck_1$ to any free variable in $ck'_1$. For example, $S \oplus (\{ck_1 < ck_2\})$ means that, in $S$, $ck_1$ never appears on the left of an inequation that leads transitively to an inequation where $ck_2$ appears on the right.

A system $S$ is satisfiable if there exists a substitution $\rho$ from variables to infinite binary words such that for all $\{ck_i < ck'_i\} \in S$, $\rho(ck_i) < \rho(ck'_i)$.

There is a straightforward but important (weak) confluence property on subsampling and satisfiability:

**PROPOSITION 12** (subsampling and satisfiability). If $\alpha' \notin S$, then for all $w$, $S$ is satisfiable if $S(\alpha' w/\alpha) \vdash w$ is satisfiable.

**Proof.** Suppose $S$ is satisfiable with $\rho(\alpha) = \gamma$ on $m$. Then we can build another substitution $\rho'$ satisfying the system of constraints by choosing $\rho'(\gamma) = \gamma'$ on $\mathcal{P}(m, w)$, $\rho'(\alpha) = \gamma'$ on $\mathcal{P}(m, w)$ on $m$ and $\rho'(\alpha') = \gamma'$ on $\mathcal{P}(m, w)$. The reciprocal is obvious. \(\square\)

Let us eventually define three functions useful to bound the subsampling of a given word: $\underline{\alpha}_{\min}$, $\underline{\alpha}_{\max}$ and $\Delta$ are defined recursively as follows:

$$\underline{\alpha}_{\min}(0^i1^b, w) = 0^i1^b, \underline{\alpha}_{\min}(w)$$
$$\underline{\alpha}_{\max}(0^i1^b, w) = 0^i1^b, \underline{\alpha}_{\max}(w)$$
$$\Delta(0^i1^b, w, r) = 1^i1^b0^r, \Delta(0^i1^b, w, r + \lfloor r + f \rfloor) = 2^i1^b, c' = c + \lfloor r + g \rfloor$$

Notice $\Delta$ — from pairs of infinite binary words and rational numbers to infinite binary words—is of technical interest for the proofs only.

**PROPOSITION 13.** For all $w$, $\underline{\alpha}_{\min}(w) \leq \underline{\alpha}_{\max}(w)$, $\underline{\alpha}_{\min}(w) :\leq \underline{\alpha}_{\max}(w)$, and $\underline{\alpha}_{\min}(w) w = \underline{\alpha}_{\max}(w)$ on $w$.

**Proof.** The first part of the proposition is proven inductively on the position of $\alpha$ in the subsampling.

The second part is a consequence of the definition of $c'$, designed to match the asymptotic rate of $\alpha$ in $u$ through the propagation of $r$, the fractional part of the asymptotic rate. \(\square\)

The set of subsamples of a given word is characterized by the following technical proposition:

**PROPOSITION 14.** For all $v, w$, we have

$$\underline{\alpha}_{\min}(w) \leq \underline{\alpha}_{\max}(w) \iff v \leq \underline{\alpha}_{\min}(w) \implies v \leq \underline{\alpha}_{\max}(w)$$

and

$$v \leq \underline{\alpha}_{\min}(w) \leq \underline{\alpha}_{\max}(w) \implies v \leq \underline{\alpha}_{\max}(w)$$

**Proof.** The first pair of implications is proven inductively on the definition of $\underline{\alpha}_{\min}$ and $\underline{\alpha}_{\max}$.

For the second pair of implications, observe that

$$1^i1^b0^r1^b1^b1^b \leq 1^i1^b0^r1^b1^b1^b$$

and

$$1^i1^b0^r1^b1^b1^b \leq 0^r1^b0^r1^b1^b1^b$$

hence $\Delta(u, w, 0)$ on $\underline{\alpha}_{\min}(w)$ (resp. $\Delta(u, w, 0)$ on $\underline{\alpha}_{\max}(w)$) yields a lower (resp. upper) bound on all $v'$ such that

$$v' \leq \Delta(u, w, 0) \implies \underline{\alpha}_{\min}(w) w = \Delta(u, w, 0) \leq \underline{\alpha}_{\max}(w) w.$$

Finally, observe that $\Delta(u, w, 0)$ is synchronous with $u$, which allows to apply the first pair of the proposition and concludes the proof.

Let us finally define the simplification relation $\sim$ between clock constraints. Its definition is given in Figure 8. Any new variable appearing in right-hand side of the simplification relation is assumed to be fresh.

**THEOREM 2** (preservation of satisfiability). If $S$ is satisfiable and $S \sim S'$ then $S'$ is satisfiable.

**Proof.** Proposition 12 authorizes to sample (to slow down) the system and will be used throughout the proof. Let us consider every relation in Figure 8.

**SUP**, **INF** and **CYCLE**. Preservation of satisfiability is a direct application of Propositions 2 and 5.

**EQUAL.** This rule preserves satisfiability: it just subsamples a pair of variables.

**CUT.** By definition of $\underline{\alpha}_{\min}$ and $\underline{\alpha}_{\max}$, the right-hand side of the relation is a sufficient condition of satisfiability.

Conversely, consider a solution $\alpha_1 = \alpha$ on $v_1$ and $\alpha_2 = \alpha$ on $v_2$.

Let $V_1 = \mathcal{P}(v_1, \underline{\alpha}_{\min}(w))$ and $V_2 = \mathcal{P}(v_1, \underline{\alpha}_{\max}(w))$, and replace $\alpha$ by $\alpha'$ on $V_1$. We have $\alpha_1 = \alpha'$ on $V_1$ and $\alpha_2 = \alpha$ on $V_2$. Let us...
choose \( \alpha_2 = \alpha' \) on \( \Delta(V_1', w, 0) \). From Proposition 14, we have
\( \alpha_1 < \alpha' \) on \( V_1' \) and \( \alpha_1 \) on \( V_{\max}(w) \). We also have \( V_1' \) on \( V_{\max}(w) \) on \( w < \cdots < V_1 \) on \( V_2 \) on \( w \), hence Proposition 14 yields \( \Delta(V_1', w, 0) \) on \( V_{\max}(w) \) on \( V_1 \) on \( V_2 \). Since \( \alpha_2 = \alpha' \) on \( V_1 \) on \( V_2 \), we have \( \alpha_1 \) on \( V_{\max}(w) \) on \( \alpha_2 \). The right-hand side of the relation is thus satisfiable.

(FORK) and (JOIN). The proof is very similar: choosing the same \( \alpha_2 \) satisfies both inequalities on \( \alpha_1 \) and \( \alpha_2 \) simultaneously.

(SUBST). Consider the form of the inequality \( I \) on \( \alpha \). The right-hand side of the relation is of course a sufficient condition of satisfiability. It is also clear that it is necessary when the inequality does not belong to a circuit. Assuming it belongs to a circuit, simplify the system through the systematic application of all other rules, enforcing that no inequality belongs to multiple simple circuits. A retiming argument [20] shows that, if the system is satisfiable, then there is a solution such that all inequalities in a given circuit but (at most) one are converted to equalities: considering a solution with at least two strict inequalities, split the circuit by renaming the common clock variable, choosing one name for the path from one inequality to the other and another one on the other path, unify any one of the broken inequalities to effectively remove this inequality from the solution.

The proof is symmetrical for the second form of \( I \).

\[ \square \]

**Rule (EQUAL)** is only provided to factor the unification step out of the (CUT), (FORK) and (JOIN) rules. As a consequence, in the following resolution algorithm, we assume rule (EQUAL) is an enabling simplification, applied once before each rule (CUT), (FORK) and (JOIN).

**THEOREM 3** (resolution algorithm). The set of rules in Figure 8 defines a non-deterministic, but always terminating resolution algorithm:

1. the tree of simplifications \( S \rightarrow S' \) is finite;
2. if \( S \) is satisfiable, there is a sequence of rule applications leading to the empty set.

**Proof.** The proof is based on the graph structure induced by \( S \). (SUP) and (INF) strictly reduces the number of acyclic paths. (EQUAL) is only used once for each application of (CUT), (FORK) and (JOIN). The \( w_1 \neq w_2 \) condition guarantees it can only be applied a finite number of times. A systematic application of (SUP), (INF), (CUT), (FORK) and (JOIN) leads to a system where no inequality belongs to multiple simple circuits. This enables (SUBST), which strictly reduces the length of any circuit or multi-path sub-graphs. (CYCLE) reduces short-circuits on a single variable.

Any ordering in the application of these rules terminates, and yields the empty set when \( S \) is satisfiable.

As a corollary:

**THEOREM 4** (completeness). For any expression \( e \), and for any period and clock environments \( P \) and \( H \), if \( e \) has an admissible clock type in \( P, H \) for the relaxed clock calculus, then the type inference algorithm computes a clock \( ct \) verifying \( P, H \vdash e : ct \).

Intuitively, if the type constraints imposed by the clock calculus are satisfiable, then our resolution algorithm discovers one solution. This strong result guarantees the clock calculus’s ability to accept all programs with periodic clocks that can be translated to a strictly \( (0-) \) synchronous framework.

Completeness would be easier to derive from principality, i.e., from the existence of a most general type for every expression [22, 1]. Yet the unification of clock stream types is not purely structural (it exploits the properties of the \( on \) operator), and there are many ways to solve an equation on clock types. There is not much hope either that the system of clock constraints can be solved by a set of confluent rules, since multiple solutions are often equivalent up to retiming [20].

Finally, although Theorem 3 proves completeness, our resolution algorithm does not guarantee anything about the quality of the result (total buffer size, period length, rate of the common clock).

**5. Translation Procedure**

When a network is associated with a system of clock inequalities where not all of them are simplified into equalities, its execution is undefined with respect to the semantics of 0-synchronous programs. Buffer processes are needed to synchronize producers with consumers.

**5.1 Translation Semantics**

Consider the input clock \( ck \) on \( w \) and the output period \( ck \) on \( w' \), with \( w \leq w' \). To fully synchronize the communication, we insert a new buffer node \( buffer_{w,w'} \) with clock \( \forall \beta \) on \( w \to \beta \) on \( w' \); \wrt \( w \) states when a push \( \text{(resp. pop)} \) occurs.

**PROPOSITION 15** (buffer size). Consider two synchronizable infinite binary words \( w \) and \( w' \) such that \( w \leq w' \). The minimal buffer to allow communication from \( w \) to \( w' \) is of size

\[
\text{size}(w, w') = \max\{\max_{p \in \mathbb{P}}(|q - p| \mid |w_q| > |w_p|), 0\}.
\]

Communication from \( w \) to \( w' \) is called \( \text{size}(w, w') \)-synchronous.

On periodic words, this size is effectively computable thanks to Remark 1.

**Proof.** This is the maximal number of pending writes appearing before their matching reads, hence a lower bound on the minimal size. It is also the minimal size, since it is possible to implement a size \( n \) buffer with \( n \) registers.

For the simplified downscaler, buffer size is equal to 1, since clock \( 0^{1000010000000} 10000000100 \) may take at most one advance tick with respect to clock \( 0^{10000000100} \). For the real downscaler, we automatically computed the size \( 880 \).

Let us now define a translation semantics for programs accepted with the relaxed clock calculus. This will enable us to state the cornerstone result of this work, namely that programs accepted with the relaxed clock calculus can be turned into synchronous programs which are accepted by the original clock calculus. This is obtained through a program transformation which inserts a buffer every time a strict inequality on stream clock types remains after resolution. Because a buffer is itself a synchronous program, the resulting translated program can be clocked with the initial system and can thus be synchronously evaluated. This translation is obtained by asserting judgment \( P, H \vdash s : ct \Rightarrow e' \), meaning that in the period environment \( P \) and the clock environment \( H \), the expression \( e \) with clock \( ct \) is translated into \( e' \). The insertion rule is:

\[ \begin{align*}
\text{(TRANSITION)} & \quad P, H \vdash s : ck \ \text{on} \ w \Rightarrow \text{buffer}_{w,w'}(e') \\
& \quad P, H \vdash s : ck \ \text{on} \ w' \Rightarrow \text{buffer}_{w',w}(e')
\end{align*} \]

Other rules are simple morphisms.\(^8\)

**5.2 Practical Buffer Implementation**

From the definition in Section 4.3.1, one may define a custom buffer process with the exact clock type to resynchronize a communication. Yet this definition suffers from the intrinsic combinatorics

\(^8\) Notice the (CTR) rule shifts a clock constraint imposed by the programmer; this rule will often lead to the insertion of a synchronization buffer, triggering the (TRANSITION) rule indirectly.
linear in the buffer size, which is appropriate for a practical implementation. The downside is that static properties about the process become much harder to exhibit for automated tools (model checking, abstract interpretation): in particular, it is hard to prove that the code actually behaves as a FIFO buffer when at most \( n \) tokens are sent and not yet received.

\[
\text{let node buffer1 (push, pop, i) = (empty, o) where o = if pempty then i else pmemo and memo = if push then i else pmemo and pmemo = 0 fby memo and empty = if push then if pop then pempty else false else if pop then true else pempty and pempty = true fby empty}
\]

**Figure 9.** A synchronous buffer

A buffer of size one, called 1-buffer, can be written as a synchronous program with three inputs and two outputs. It has two boolean inputs push and pop and a data i, o and empty are the outputs. Its behavior is the following: the output o equal i when its internal memory was empty and equals the internal memory otherwise. Then, the memory is set to i when push is true. Finally, the empty flag gives the status of the internal memory. If a push and a pop occur and the memory is empty, then the buffer is bypassed. If a push occurs only, empty becomes false. Conversely, if a pop occurs then the memory is emptied. This behavior can be programmed in a synchronous language. Figure 10 gives an implementation of this buffer in a strictly synchronous language.\(^9\) Buffers of size \( n \) can be constructed by connecting a sequence of 1-buffers as shown in Figure 9. To complete these figures, notice the boolean streams push and pop need to be computed explicitly from the periodic words \( w \) and \( w' \) of the output and input stream clocks.

Finally, because safety is already guaranteed by the calculus on periodic clocks, a synchronous implementation for the buffer is not absolutely required. An array in random-access memory with head and tail pointers would be correct by construction, as soon as it satisfies the size requirements.

### 5.3 Correctness

We define judgment \( P, H \vdash e : ct \) to denote that expression \( e \) has clock \( c \) in the period environment \( P \) and the clock environment \( H \), for the original 0-synchronous system. The following result states that any program accepted by the relaxed clock calculus translates to an equivalent 0-synchronous program (in terms of data-flow on streams). This equivalent program has the same clock types.

**Theorem 5** (correctness). For any period environment \( P \) and clock environment \( H \), if \( P, H \vdash e : ct \Rightarrow e' : ct \).

The proof derives from the subtyping rule underlying \( \vdash \) judgments: classical subtyping theory \([22,1,23]\) reduces global correctness to the proof of local 0-synchronism of each process composition in the translated program (including at clock constraints). This is guaranteed by the previous buffer insertion scheme, since each buffer’s signature is tailored to the resynchronization of a pair of different but synchronizable clocks. This ensures the translated program is synchronous.

### 6. Synchrony and Asynchrony

A system that does not have a single synchronous clock is not necessarily asynchronous: numerous studies have tackled with relaxed or multi-clocked synchrony at the hardware or software levels. We only discuss the most closely related studies, a wide and historical perspective can be found in \([7]\).

There are a number of approaches to the specification and design of hybrid hardware/software systems. Most of them are graphical tools based on process networks. Kahn process networks (KPN) \([18]\) is a fundamental one, but it models only functional properties, as opposed to structural properties. KPN are used in a number of tools such as YAPI \([14]\) or the COSY project \([5]\); such tools still require expertise from different domains and there is no universal language that combines functional and structural features in a single framework.

Steps towards the synchronous control of asynchronous systems are also conducted in the domain of synchronous programming languages, such as the work of Le Guernic et al. \([19]\) on Polychrony. This work targets the automatic and correct by construction refinement of programs, in the same spirit as our clock composition, but it does not consider quantitative properties of clocks. StreamIt \([24]\) is a language for high performance streaming computations that tackles mainly stream-level and algebraic optimization issues.

Ptolemy \([6]\) is a rich platform with simulation and analysis tools for the design of embedded streaming systems: it is based on the synchronous data-flow (SDF) model of computation \([15]\). Unlike synchronous languages, SDF graphs cannot express (bounded or not) recursion and arbitrary aperiodic execution. They are not explicitly clocked either: synchrony is a consequence of local balance equations on periodic execution schemes. The SDF model allows static scheduling and is convenient for the automatic derivation of timing properties \([21]\), but the lack of clocks weakens its amenability for formal reasoning and correct-by-construction generation of synchronous code, with respect to synchronous languages \([17,2]\). Interestingly, \(n\)-synchronizable clocks seem to fill this hole, leading to the definition of a formal semantics for SDF while exposing the

---

precise static schedule to the programmer (for increased control on buffer management and code generation). Further analyses of the correspondence between the two models are left for future work.

7. Conclusion and Perspectives

We proposed a synchronous programming language to implement concurrency-by-construction, high-performance streaming applications. Our model addresses the automatic synthesis of communications between processes that are not strictly synchronous. In this model, we show that latencies and buffer requirements can be inferred automatically. We extend a core data-flow language with a notion of periodic clocks and with a relaxed clock calculus to compose synchronous processes. This relaxed synchronous model defines a formal semantics for synchronous data-flow graphs, building a long awaited bridge with synchronous languages. The clock calculus and the translation procedure from relaxed synchronous to strictly synchronous programs are proven correct, and the associated type inference is proven complete. An implementation in the synchronous language LUCID SYNCHRON is under way and was applied to a classical video downscaler example. We believe this work widens the scope of synchronous programming beyond safety-critical reactive systems and circuit synthesis, promising increased safety and productivity in the design and optimization of a large spectrum of applications.

Acknowledgments

This work is partially funded by the French “ACI Sécurité” Alidecs. We are grateful to anonymous reviewers for their careful reading and deep remarks which contributed to improve the final version of this paper. We would also like to thank zbigniew Chamski who contributed several key ideas that lead to the concept of n-synchrony.

References