N-Synchronous Kahn Networks

A Relaxed Model of Synchrony for Real-Time Systems

Albert Cohen [1]
Albert.Cohen@inria.fr

Marc Duranton [3]
Marc.Duranton@philips.com

Christine Eisenbeis [1]
Christine.Eisenbeis@inria.fr

Claire Pagetti [1,4]
Claire.Pagetti@cert.fr

Florence Plateau [2]
Florence.Plateau@lri.fr

Marc Pouzet [2]
Marc.Pouzet@lri.fr

[1] ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud 11 University, France
[2] LRI, Paris-Sud 11 University, France
[3] Philips Research Laboratories, Eindhoven, The Netherlands
[4] CERT, ONERA, Toulouse, France

Abstract

The design of high-performance stream-processing systems is a fast growing domain, driven by markets such like 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 insurance 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 correct-by-construction properties to be proven by the compiler. These properties include resource requirements and delays between input and output streams. Automating this task avoids tedious and error-prone engineering, due to the combinatorics of the composition of filters with multiple data rates and formats. Correctness of the implementation is also difficult to assess with traditional (asynchronous, simulation-based) approaches. This language is thus provided with a relaxed notion of synchronous composition, called *n-synchrony*: two processes are n-synchronous if they can communicate in the ordinary (0-)synchronous model with a FIFO buffer of size n.

Technically, we extend a core synchronous data-flow language with a notion of periodic clocks, and design a relaxed clock calculus (a type system for clocks) to allow non strictly synchronous processes to be composed or correlated. This relaxation is associated with two sub-typing rules in the clock calculus. Delay, buffer insertion and control code for these buffers are automatically inferred from the clock types through a systematic transformation into a standard synchronous program. We formally define the semantics of the language and prove the soundness and completeness of its clock calculus and synchronization transformation. Finally, the language is compared with existing formalisms.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Algorithms, Languages, Security, Theory

Keywords Correctness by Construction, Resource Constraints, Streaming Applications, Subtyping, Synchronous Languages

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.

1. Introduction

The rapid evolution of embedded system technology, favored by Moore's law and standards, is increasingly blurring the barriers between the design of safety-critical, real-time and high-performance systems. A good example is the domain of high-end video applications, where tera-operations per second (on pixel components) in hard real-time will soon be common in low-power devices.

Unfortunately, general-purpose architectures and compilers are not suitable for the design of real-time *and* high-performance (massively parallel) *and* low-power *and programmable* system-on-chip [9]. Achieving a high compute density and still preserving programmability is a challenge for the choice of an appropriate architecture, programming language and compiler. Typically, thousands of operations per cycle must be sustained on chip, exploiting multiple levels of parallelism in the compute kernel while enforcing strong real-time properties.

Synchronous Languages Can Help To address these challenges, we studied the synchronous model of computation [2] which allows for the generation of custom, parallel hardware and software systems with *correct-by-construction structural properties*, including real-time and resource constraints. This model met industrial success for safety-critical, reactive systems, through languages like SIGNAL [3], LUSTRE (SCADE) [17] and ESTEREL [4].

To enforce real-time and resources properties, synchronous languages assume a common clock for all registers, and an overall predictable execution layer where communications and computations can be proven to take less than a (physical or logical) clock cycle. Due to wire delays, a massively parallel system-on-chip has to be divided into multiple, asynchronous clock domains: the so called *Globally Asynchronous Locally Synchronous* (GALS) model [10]. This has a strong impact on the formalization of synchronous execution itself and on the associated compilation strategies [19].

Due to the complexity of high-performance applications and to the intrinsic combinatorics of synchronous execution, our earlier work [11] showed that *multiple clock domains* have to be considered *at the application level as well*. This is the case for modular designs with separate compilation phases, and for a single 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 with bounded, statically computable buffer sizes. This extension allows the modular composition of independently scheduled components with multiple periodic clocks satisfying a flow preservation equation, through the automatic inference of bounded delays and FIFO buffers.

Main Contributions More technically, we define a relaxed clock-equivalence principle, called n-synchrony. A given clock ck_1 is *n-synchronizable* with another clock ck_2 if there exists a data-flow (causality) preserving way of making ck_1 synchronous with ck_2 applying a constant delay to ck_2 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 ck_1 nor ck_2 are modified (besides the optional insertion of a constant delay); schedule choices associated with ck_1 and ck_2 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 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.

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 process composition through translation to a strictly synchronous program, by automatically inserting buffers with minimal size. Section 6 discusses related work at the frontier between synchronous and asynchronous systems. We conclude in Section 7.

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 downscaler [9], depicted in Figure 1. It converts a high definition (HD) video signal,

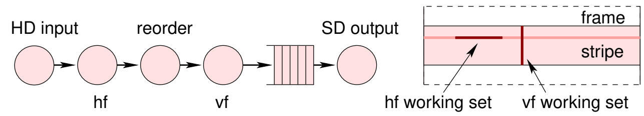

Figure 1. The downscaler

1920×1080 pixels per frame, into a standard definition (SD) output for TV screen, that is 720×480:[1]

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 (o_t) by interpolating 6 consecutive pixels (p_j) weighted by coefficients given in a predetermined matrix (example of a 64 phases, 6-taps polyphase filter [9]):

$$o_t = \sum_{k=0}^{5} p_{t\times 1920/720+k} \times \mathrm{coef}(k, t \bmod 64).$$

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

At every clock tick, the `hf` function computes the interpolation of six consecutive pixels of the input `p` (`0 fby p` stands for the previous value of `p` initialised with value `0`). The implementation of `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

[1] Here we only consider the active pixels for the ATSC or BS-Digital High Definition standards.

```
let clock c = ok where rec
  cnt = 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
  o2 = 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`

stream `c` used as a clock to sample the output of the horizontal filter. The `let clock` construction identifies syntactically these particular boolean streams. Here is a possible execution diagram.

`c`	*true*	*false*	*true*	*false*	*false*	*true*	*false*	...
`p`	3	4	7	5	6	10	12	...
`o2`	0	3	4	7	5	6	10	...
`o3`	0	0	3	4	7	5	6	...
`o4`	0	0	0	3	4	7	5	...
`o5`	0	0	0	0	3	4	7	...
`o6`	0	0	0	0	0	3	4	...
`o`	3		14			35		...

In the synchronous data-flow model, each variable/expression 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 *false* stands for the absence and *true* for the presence. E.g., if `x` is an integer stream variable, then `x+1` and `x` 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 `hf` is $\forall\alpha.\alpha \to \alpha$ `on` c (printed `'a -> 'a on c`) meaning that for any clock α, if input `p` has clock α, then the output is on a subclock α `on` c defined by the instant where the boolean condition c is *true*.

In synchronous languages, clock conditions such as `c` can be arbitrarily complex boolean expressions, meaning that compilers make no hypothesis on them. Yet the applications we consider have a periodic behavior; thus a first simplification consists in enhancing the syntax and semantics with the notion of *periodic clocks*.

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., $192.915\mu s$, 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 outgoing 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:

$$\begin{array}{lcl} w & ::= & u(v) \\ u & ::= & \varepsilon \mid 0 \mid 1 \mid 0.u \mid 1.u \\ v & ::= & 0 \mid 1 \mid 0.v \mid 1.v \end{array}$$

where $(v) = \lim_n v^n$ 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 $\mathbb{Q}_2 - \mathbb{N}$.

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 (01) and to $01(01)$. Fortunately, there exists a normal representation: it is the unique representation of the form $u(v)$ with the shortest prefix *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., $(10) \preceq (01) \preceq 0(01) \preceq (001)$. 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 $w \sqcup w'$ and lower bound $w \sqcap w'$ of two infinite binary words with

$$\begin{array}{l} \forall p \geq 1, [w \sqcup w']_p = \max([w]_p, [w']_p) \\ \forall p \geq 1, [w \sqcap w']_p = \min([w]_p, [w']_p). \end{array}$$

E.g., $1(10) \sqcup (01) = (01)$ and $1(10) \sqcap (01) = 1(10)$; $(1001) \sqcup (0110) = (01)$ and $(1001) \sqcap (0110) = (10)$.

PROPOSITION 1. *The set* $\big((0+1)^\omega, \preceq, \sqcup, \sqcap, \bot = (1), \top = (0)\big)$ *is a complete lattice.*

Notice $\top$ is indeed (0) since $[(0)]_p = \infty$ for all $p > 0$.[2]

[2] Yet the restriction of this lattice to $\mathbb{Q}_2$ is *not* complete, neither upwards nor downwards, even within $\mathbb{Q}_2 - \mathbb{N}$.

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.*

1. *One may choose a, a', b, and b' with $|a| = |a'| = max(|u|,|u'|)$ and $|b| = |b'| = lcm(|v|,|v'|)$ where lcm stands for* least common multiple. *Indeed, assuming $|u| \leq |u'|$, $p = |u'| - |u|$ and $n = lcm(|v|,|v'|)$: $w = u.v[1]\dots v[p].\big((v[p+1]\dots v[p+|v|])^{n/|v|}\big)$ and $w' = u'(v'^{n/|v'|})$. E.g., words* $010(001100)$ *and* $10001(10)$ *can be rewritten into* $01000(110000)$ *and* $10001(101010)$.
2. *Likewise, one may obtain prefixes and suffixes with the same number of 1s: $w = a(b)$ and $w' = a'(b')$ with $|a|_1 = |a'|_1 = max(|u|_1,|u'|_1)$ and $|b|_1 = |b'|_1 = lcm(|v|_1,|v'|_1)$. Indeed, suppose $|u|_1 \leq |u'|_1$, $|v|_1 \leq |v'|_1$, $p = |u'|_1 - |u|_1$, $r = [v]_p$, and $n = lcm(|v|_1,|v'|_1)$: $w = u.v[1]\dots v[r].\big((v[r+1]\dots v[r+|v|])^{n/|v|_1}\big)$ and $w' = u'.(v'^{n/|v'|_1})$. E.g., the pair of words* $010(001100)$ *and* $10001(10)$ *become* $010001(100001)$ *and* $10001(1010)$.
3. *Finally, one may write $w = a(b)$ and $w' = a'(b')$ with $|a|_1 = |a'|$ and $|b|_1 = |b'|$. Indeed, suppose $|u|_1 \leq |u'|$, $|v|_1 \leq |v'|$, $p = |u'|_1 - |u|$, $r = [v]_p$, $n = lcm(|v|_1,|v'|)$: $w = u.v[1]\dots v[r].\big((v[r+1]\dots v[r+|v|])^{n/|v|_1}\big)$ and $w' = u'(v'^{n/|v'|})$. E.g., the pair of words* $010(001100)$ *and* $10001(10)$ *can be rewritten into* $0100011000011(000011)$ *and* $10001(10)$.

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:

$$c ::= w \mid c \text{ on } w, \quad w \in \{0,1\}^{\omega}.$$

If c is a clock and w is an infinite binary word, then c on w denotes a *subsampled clock* of c, where w is itself set on clock c. In other words, c on w is the clock obtained in advancing in clock w at the pace of clock c. E.g., (01) on $(101) = (010101)$ on $(101) = (010001)$.

c	0	1	0	1	0	1	0	1	0	1	...	(01)
w		1		0		1		1		0	...	(101)
c on w	0	1	0	0	0	1	0	1	0	0	...	(010001)

Formally, on is inductively defined as follows:

$$\begin{array}{rcl} 0.w \text{ on } w' & = & 0.(w \text{ on } w') \\ 1.w \text{ on } 0.w' & = & 0.(w \text{ on } w') \\ 1.w \text{ on } 1.w' & = & 1.(w \text{ on } w') \end{array}$$

Clearly, the on operator is *not* commutative.

PROPOSITION 2. *Given two infinite binary words w and w', the infinite binary word w on w' satisfies the equation $[w \text{ on } w']_p = [w]_{[w']_p}$ for all $p \geq 1$.*

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

There is an important corollary:

PROPOSITION 3 (on-associativity). *Let w_1, w_2 and w_3 be three infinite binary words.*

Then w_1 on $(w_2$ on $w_3) = (w_1$ on $w_2)$ on w_3.

Indeed $[w_1 \text{ on } w_2]_{[w_3]_p} = [w_1]_{[w_2]_{[w_3]_p}} = [w_1]_{[w_2 \text{ on } w_3]_p}$.

The following properties also derive from Proposition 2:

PROPOSITION 4 (on-distributivity). *the* on *operator is distributive with respect to the lattice operations $\sqcap$ and $\sqcup$.*

PROPOSITION 5 (on-monotonicity). *For any given infinite binary word w, functions $x \mapsto x$ on w 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:

$$c ::= w \mid c \text{ on } w, \quad w \in \mathbb{Q}_2.$$

In the case of these periodic clocks, proposition 2 becomes an algorithm, allowing to effectively compute the result of c on w. Let us consider two infinite periodic binary words $w_1 = u_1(v_1)$ and $w_2 = u_2(v_2)$ with $|u_1|_1 = |u_2|$ and $|v_1|_1 = |v_2|$, this is possible because of Remark 1. Then $w_3 = w_1$ on $w_2 = u_3(v_3)$ is computed by $|u_3| = |u_1|$, $|u_3|_1 = |u_2|_1$, $[u_3]_p = [u_1]_{[u_2]_p}$ and $|v_3| = |v_1|$, $|v_3|_1 = |v_2|_1$, $[v_3]_p = [v_1]_{[v_2]_p}$.

Likewise, periodic clocks are closed for the pointwise extensions of boolean operators `or`, `not`, and `&`.

3.3 Synchronizability

Motivated by the downscaler 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 $w \bowtie w'$, iff there exists $d,d' \in \mathbb{N}$ such that $w' \preceq 0^d w'$ and $w' \preceq 0^{d'} w$. It means that we can delay w by d' ticks so that the 1s of w' occur before the 1s 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 (0) are not synchronizable; (010) 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 $w \bowtie w'$, iff they have the same* rate *(a.k.a.* throughput*)*

$$|v|_1/|v| = |v'|_1/|v'|.$$

In other words, $w \bowtie w'$ 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 $|v| = |v'|$. $w_1 = u(v) \bowtie w_2 = u'(v')$ iff there exists d,d' s.t. $\forall w \leq w_2[1..|u|+|v|+d]$, $w' \leq 0^d.w_2[1..|u|+|v|] \wedge |w| = |w'| \implies |w|_1 \geq |w'|_1$ and $\forall w \leq 0^{d'} w_1[1..|u|+|v|]$, $w' \leq w_2[1..|u|+|v|+d'] \wedge |w| = |w'| \implies |w|_1 \geq |w'|_1$. It is sufficient to cover the prefixes of finite length $\leq |u|+|v|+max(d+d')$.*

Case $|v'|_1 = 0$ is straightforward. Let us assume that $|v|_1/|v'|_1 > |v|/|v'|$ (the case $|v|_1/|v'|_1 < |v|/|v'|$ is symmetric). Because of Remark 1, it means $|v|_1/|v'|_1 > 1$. Then it entails that (v) and (v') are not synchronizable so as w_1 and w_2. Let us denote $a = |v|_1 - |v'|_1$, then v^n has na 1 more than v'^n. Thus $v^n \preceq 0^{f(n)} v'^n$ where $|v^n| \geq f(n) \geq na$ and $f(n)$ is minimal in the sense that $v^n \not\preceq 0^{f(n)-1} v'^n$. It entails that $(v) \preceq 0^{lim f(n)}(v')$ and thus there are not synchronizable.

[3] E.g., (1001) on $(10) = (1100)$ on (10).

Conversely, assume $|v|_1/|v'|_1 = |v|/|v'|$. Since u and u' are finite, we have $1^r u \preceq 0^p u'$ and $1^k u' \preceq 0^q u$ with $r = max(0, |u'|_1 - |u|_1)$, $k = max(0, |u|_1 - |u'|_1)$. (v), $p = \min\{l \mid l \leq |u| + r \wedge 1^r u \preceq 0^l u'\}$ and $q = \min\{l \mid l \leq |u'| + \wedge 1^k u' \preceq 0^l u\}$. (v') are also synchronizable, thus $(v) \preceq 0^m(v')$ and $(v') \preceq 0^n(v)$. Then $w_1 \preceq 0^{p+m+r|v|} w_2$ and $w_2 \preceq 0^{q+n+k|v'|} w_1$. 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` $x = e$),[4] applications ($e(e)$), initialized delays (e `fby` 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; `fst` and `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 (`let node` $f\ x = e$) and periodic clocks (`period` $p = pe$). E.g., `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 `period` $quarter = half$ `on` $half$. Periodic clocks can be combined with boolean operators. Note that clocks are *static* expressions which can be simplified at compile time into the normal form $u(v)$ of infinite periodic binary words.

$$\begin{array}{lcl} e & ::= & x \mid i \mid (e,e) \mid e \texttt{ where } x = e \mid e(e) \mid op(e,e) \\ & & \mid e \texttt{ fby } e \mid e \texttt{ when } pe \mid \texttt{merge } pe\ e\ e \\ & & \mid \texttt{fst } e \mid \texttt{snd } e \mid e \texttt{ at } e \\ d & ::= & \texttt{let node } f\ x = e \mid d;d \\ dp & ::= & \texttt{period } p = pe \mid dp;dp \\ pe & ::= & p \mid w \mid pe \texttt{ on } pe \mid \texttt{not } pe \mid pe \texttt{ or } pe \mid pe \texttt{ \& } pe \end{array}$$

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 (100000))` is a constraint given by the programmer; it states that the output pixel `o` must be produced at some clock α `on` (100000), thus 6 times slower than the input clock α.

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].

[4] Corresponds to `let x = e in e` in ML.

```
let period c = (10100100)
let node hf p = o where rec (...)
  and o = f (p,o2,o3,o4,o5,o6) when c

let node main i = o at (i when (100000)) where rec
  t = hf i
  and (i1,i2,i3,i4,i5,i6) = reorder t
  and o = vf (i1,i2,i3,i4,i5,i6)
```

Figure 3. Synchronous code using periodic clock

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 (σ) quantified over a set of clock variables (α) 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 (α).

$$\begin{array}{lcl} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_m . ct \\ ct & ::= & ct \to ct \mid ct \times ct \mid ck \\ ck & ::= & ck \texttt{ on } pe \mid \alpha \\ H & ::= & [x_1 : \sigma_1, \ldots, x_m : \sigma_m] \\ P & ::= & [p_1 : pe_1, \ldots, p_n : pe_n] \end{array}$$

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 filter of the downscaler has clock scheme $\forall \alpha . \alpha \to \alpha$ `on` (10100100); this means that, if the input has any clock α, then the output has some clock α `on` (10100100). This clock type can in turn be instan-

[5] We shall sometimes say *clock* instead of *clock type* when clear from context.

tiated in several ways, replacing α by more precise stream clock type (e.g., some sampled clock α' on (01)).

The rules for instantiating and generalizing a clock type are given below. $FV(ct)$ denotes the set of free clock variables in ct.

$$\begin{array}{lcl} ct'[\vec{ck}/\vec{\alpha}] & \leq & \forall\vec{\alpha}.ct' \\ fgen(ct) & = & \forall\alpha_1,...,\alpha_m.ct \text{ where } \alpha_1,...,\alpha_m = FV(ct) \end{array}$$

It states that a clock scheme can be instantiated by replacing variables with clock expressions; $fgen(ct)$ 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.

$$\text{(IM)}\quad P,H \vdash i : ck$$

$$\text{(INST)}\quad \frac{ct \leq H(x)}{P,H \vdash x : ct}$$

$$\text{(OP)}\quad \frac{P,H \vdash e_1 : ck \quad P,H \vdash e_2 : ck}{P,H \vdash op(e_1,e_2) : ck}$$

$$\text{(FBY)}\quad \frac{P,H \vdash e_1 : ck \quad P,H \vdash e_2 : ck}{P,H \vdash e_1 \texttt{ fby } e_2 : ck}$$

$$\text{(WHEN)}\quad \frac{P,H \vdash e : ck \quad P \vdash pe}{P,H \vdash e \texttt{ when } pe : ck \texttt{ on } pe}$$

$$\text{(MERGE)}\quad \frac{P \vdash pe \quad H \vdash e_1 : ck \texttt{ on } pe \quad P,H \vdash e_2 : ck \texttt{ on not } pe}{P,H \vdash \texttt{merge } pe\ e_1\ e_2 : ck}$$

$$\text{(APP)}\quad \frac{P,H \vdash e_1 : ct_2 \rightarrow ct_1 \quad P,H \vdash e_2 : ct_2}{P,H \vdash e_1(e_2) : ct_1}$$

$$\text{(WHERE)}\quad \frac{P,H,x : ct \vdash e_1 : ct_1 \quad P,\ H,x : ct \vdash e_2 : ct_2}{P,H \vdash e_2 \texttt{ where } x = e_1 : ct_2}$$

$$\text{(PAIR)}\quad \frac{P,H \vdash e_1 : ct_1 \quad P,H \vdash e_2 : ct_2}{P,H \vdash (e_1,e_2) : ct_1 \times ct_2}$$

$$\text{(FST)}\quad \frac{P,H \vdash e : ct_1 \times ct_2}{P,H \vdash \texttt{fst } e : ct_1}$$

$$\text{(SND)}\quad \frac{P,H \vdash e : ct_1 \times ct_2}{P,H \vdash \texttt{snd } e : ct_2}$$

$$\text{(CTR)}\quad \frac{P,H \vdash e_1 : ck \quad P,H \vdash e_2 : ck}{P,H \vdash e_2 \texttt{ at } e_1 : ck}$$

$$\text{(NODE)}\quad \frac{P,\ H,x : ct_1 \vdash e : ct_2}{H \vdash \texttt{let node } f\ x = e : [f : fgen(ct_1 \rightarrow ct_2)]}$$

$$\text{(PERIOD)}\quad \frac{P \vdash pe}{P \vdash \texttt{period } p = pe : [p : pe]}$$

$$\text{(DEFH)}\quad \frac{H \vdash dh_1 : H_1 \quad H,H_1 \vdash dh_2 : H_2}{H \vdash dh_1;dh_2 : H_1,H_2}$$

$$\text{(DEFP)}\quad \frac{P \vdash dp_1 : P_1 \quad P,P_1 \vdash dp_2 : P_2}{P \vdash dp_1;dp_2 : P_1,P_2}$$

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 fby 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 clock 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.[6]
- Rules (PERIOD), (DEFH) and (DEFP) 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 SYNCHRONE, 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_2 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 $(\alpha \text{ on } (01)) \text{ on } (01)$ and $\alpha \text{ on } (0001)$ 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 $\perp$ representing the absence of a value.

We need a few preliminary notations. If T is a set, T^∞ denotes the set of finite or infinite sequences of elements over the set T ($T^\infty = T^* + T^\omega$). The empty sequence is noted ε and $x.s$ denotes the sequence whose head is x and tail is s. Let $\leq$ be the prefix order over sequences, i.e., $x \leq y$ if x is a prefix of y. The ordered set $D = (T^\infty, \leq)$ 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. $[D_1 \rightarrow D_2]$ 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

[6] 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.

mapping from D_1 to D_2, we shall write $fix(f) = lim_{n\to\infty} f^n(\varepsilon)$ for the smallest fix point of f (Kleene theorem). We define the set *ClockedStream*(T) of *clocked sequences* as the set of finite and infinite sequences of elements over the set $T_\perp = T \cup \{\perp\}$.

$$\begin{array}{lcl} T_\perp & = & T \cup \{\perp\} \\ \mathit{ClockedStream}(T) & = & (T_\perp)^\infty \end{array}$$

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 *clock* from clocked sequences to boolean sequences:

$$\begin{array}{lcl} clock(\varepsilon) & = & \varepsilon \\ clock(\perp.s) & = & 0.clock(s) \\ clock(x.s) & = & 1.clock(s) \end{array}$$

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 $\perp.s$ 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., `fby`) from its interpretation as a stream transformer.

$$\begin{array}{lcl}
\mathtt{const}^\#\ i\ 1.c & = & i.\mathtt{const}^\#\ i\ c \\
\mathtt{const}^\#\ i\ 0.c & = & \perp.\mathtt{const}^\#\ i\ c \\
op^\#(s_1,s_2) & = & \varepsilon \text{ if } s_1=\varepsilon \text{ or } s_2=\varepsilon \\
op^\#(\perp.s_1,\perp.s_2) & = & \perp.op^\#(s_1,s_2) \\
op^\#(v_1.s_1,v_2.s_2) & = & (v_1\, op\, v_2).op^\#(s_1,s_2) \\
\mathtt{fby}^\#(\varepsilon,s) & = & \varepsilon \\
\mathtt{fby}^\#(\perp.s_1,\perp.s_2) & = & \perp.\mathtt{fby}^\#(s_1,s_2) \\
\mathtt{fby}^\#(v_1.s_1,v_2.s_2) & = & v_1.\mathtt{fby1}^\#(v_2,s_1,s_2) \\
\mathtt{fby1}^\#(v,\varepsilon,s) & = & \varepsilon \\
\mathtt{fby1}^\#(v,\perp.s_1,\perp.s_2) & = & \perp.\mathtt{fby1}^\#(v,s_1,s_2) \\
\mathtt{fby1}^\#(v,v_1.s_1,v_2.s_2) & = & v.\mathtt{fby1}^\#(v_2,s_1,s_2) \\
\mathtt{when}^\#(\varepsilon,c) & = & \varepsilon \\
\mathtt{when}^\#(\perp.s,c) & = & \perp.\mathtt{when}^\#(s,c) \\
\mathtt{when}^\#(v.s,1.c) & = & v.\mathtt{when}^\#(s,c) \\
\mathtt{when}^\#(v.s,0.c) & = & \perp.\mathtt{when}^\#(s,c) \\
\mathtt{merge}^\#(c,s_1,s_2) & = & \varepsilon \text{ if } s_1=\varepsilon \text{ or } s_2=\varepsilon \\
\mathtt{merge}^\#(1.c,v.s_1,\perp.s_2) & = & v.\mathtt{merge}^\#(c,s_1,s_2) \\
\mathtt{merge}^\#(0.c,\perp.s_1,v.s_2) & = & v.\mathtt{merge}^\#(c,s_1,s_2) \\
\mathtt{not}^\#1.c & = & 0.\mathtt{not}^\#c \\
\mathtt{not}^\#0.c & = & 1.\mathtt{not}^\#c \\
\mathtt{on}^\#(1.c_1,1.c_2) & = & 1.\mathtt{on}^\#(c_1,c_2) \\
\mathtt{on}^\#(1.c_1,0.c_2) & = & 0.\mathtt{on}^\#(c_1,c_2) \\
\mathtt{on}^\#(0.c_1,c_2) & = & 0.\mathtt{on}^\#(c_1,c_2)
\end{array}$$

Figure 5. Semantics for the core primitives

- The `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, $\mathtt{const}^\#\ i\ c$ denotes a constant stream with stream clock c ($clock(\mathtt{const}^\#\ i\ c) = c$).
- 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).
- `fby` is the unitary delay: it "conses" the head of its first argument to its second one. The arguments and result of `fby` must be on the same clock. `fby` corresponds to a two-state machine: while both arguments are absent, it emits nothing and stays in its initial state ($\mathtt{fby}^\#$). When both are present, it emits its first argument and enters the new state ($\mathtt{fby1}^\#$) storing the previous value of its second argument. In this state, it emits a value every time its two arguments are present.
- The sampling operator expects two arguments on the same clock. The clock of the result depends on the boolean condition (c).
- The definition of `merge` states that one branch must be present when the other is absent.
- Note that $\mathtt{not}^\#$ and $\mathtt{on}^\#$ 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 ($\vdash$ judgments). We define the interpretation of clock types as the following:

$$\begin{array}{lcl}
[\![ct_1 \to ct_2]\!]_P & = & [[\![ct_1]\!]_P \to [\![ct_2]\!]_P] \\
[\![ct_1 \times ct_2]\!]_P & = & [\![ct_1]\!]_P \times [\![ct_2]\!]_P \\
s \in [\![\forall\alpha_1,\ldots,\alpha_n.ct]\!]_P & = & \forall ck_1,\ldots,ck_n, \\
 & & \quad s \in [\![ct[ck_1/\alpha_1,\ldots,ck_n/\alpha_n]]\!]_P \\
s \in [\![ck]\!]_P & = & clock(s) \leq P(ck)
\end{array}$$

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 ck (and in particular the empty sequence ε). This way, an equation $x = x+1$ which is well clocked (since $P,H,x : ck \vdash x+1 : ck$) but not causal (its smallest solution is ε) can receive a synchronous semantics.

For any period environment P, clock environment H and any assignment ρ (which maps variable names to values) such that $\rho(x) \in [\![H(x)]\!]_P$, the meaning of an expression is given by $[\![P,H \vdash e : ct]\!]_\rho$ such that $[\![P,H \vdash e : ct]\!]_\rho \in [\![ct]\!]_P$. 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 α_1.
2. The horizontal filter has the following signature, corresponding to the effective synchronous implementation of the process: $\alpha_2 \to \alpha_2 \text{ on } (10100100)$.
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 proess the clock signature $\alpha_3 \to \alpha_3 \text{ on } 0^{3600}(1)$.
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:

   $$\alpha_4 \ \to\ \alpha_4 \text{ on } (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720})$$

   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: $\alpha_4 \to \alpha_4 \text{ on } (101001001)$.
5. Finally, the designer has required that if the global input `i` is on clock α_1, then the clock of the output `o` should be $\alpha_1 \text{ on } (100000)$ — the 6 times subsampled input clock — tol-

$$
\begin{array}{rcl}
[\![P,H \vdash op(e_1,e_2) : ck]\!]_\rho & = & op^\#([\![P,H \vdash e_1 : ck]\!]_\rho, [\![P,H \vdash e_2 : ck]\!]_\rho) \\
[\![P,H \vdash x : ct]\!]_\rho & = & \rho(x) \\
[\![P,H \vdash i : ck]\!]_\rho & = & i^\#[\![ck]\!]_P \\
[\![P,H \vdash e_1 \texttt{ fby } e_2 : ck]\!]_\rho & = & \texttt{fby}^\#([\![P,H \vdash e_1 : ck]\!]_\rho, [\![P,H \vdash e_2 : ck]\!]_\rho) \\
[\![P,H \vdash e \texttt{ when } pe : ck \texttt{ on } pe]\!]_\rho & = & \texttt{when}^\#([\![P,H \vdash e : ck]\!]_\rho, P(pe)) \\
[\![P,H \vdash \texttt{merge } pe\ e_1\ e_2 : ck]\!]_\rho & = & \texttt{merge}^\#(P(pe), [\![P,H \vdash e_1 : ck \texttt{ on } pe]\!]_\rho, [\![P,H \vdash e_2 : ck \texttt{ on not } pe]\!]_\rho) \\
[\![P,H \vdash e_1(e_2) : ct_2]\!]_\rho & = & ([\![P,H \vdash e_1 : ct_1 \to ct_2]\!]_\rho)([\![P,H \vdash e_2 : ct_1]\!]_\rho) \\
[\![P,H \vdash e_1, e_2 : ct_1 \times ct_2]\!]_\rho & = & ([\![P,H \vdash e_1 : ct_1]\!]_\rho, [\![P,H \vdash e_2 : ct_2]\!]_\rho) \\
[\![P,H \vdash \texttt{fst } s_1, s_2 : ct_1]\!]_\rho & = & s_1 \text{ where } s_1, s_2 = [\![P,H \vdash e : ct_1 \times ct_2]\!]_\rho \\
[\![P,H \vdash \texttt{snd } s_1, s_2 : ct_2]\!]_\rho & = & s_2 \text{ where } s_1, s_2 = [\![P,H \vdash e : ct_1 \times ct_2]\!]_\rho \\
[\![P,H \vdash e' \texttt{ where } x = e : ct']\!]_\rho & = & [\![P,\ H, x : ct \vdash e' : ct']\!]_{\rho[x^\infty/x]} \text{ where } x^\infty = \mathit{fix}(d \mapsto [\![P,\ H, x : ct \vdash e : ct]\!]_{\rho[d/x]}) \\
[\![P,H \vdash \texttt{let node } f(x) = e : \mathit{fgen}(ct_1 \to ct_2)]\!]_\rho & = & [(d \mapsto [\![P,\ H, x : ct_1 \vdash e : ct_2]\!]_{\rho[d/x]})/f] \\
[\![P,H \vdash e_1 \texttt{ at } e_2 : ck]\!]_\rho & = & [\![P,H \vdash e_1 : ck]\!]_\rho
\end{array}
$$

Figure 6. Data-flow semantics over clocked sequences

erating an additional delay that must automatically be deduced from the clock calculus.

The composition of all 5 processes yield the type constraints $\alpha_1 = \alpha_2, \alpha_3 = \alpha_2 \texttt{ on } (10100100)$, and $\alpha_4 = \alpha_3 \texttt{ on } 0^{3600}(1)$. Finally, after replacing variables by their definitions, we get for the output `o` the following clock type:

$$((\alpha_1 \texttt{ on } (10100100)) \texttt{ on } 0^{3600}(1)) \texttt{ on } (101001001) = \\ \alpha_1 \texttt{ on } 0^{9600}(1000010000000010000000100).$$

Yet, the result is *not* equal to the clock constraint stating that `o` should have clock type $\alpha_1 \texttt{ on } (100000)$. 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 $ck \texttt{ on } w$ and $ck \texttt{ on } w'$ are* synchronizable*, and we write $ck \texttt{ on } w \bowtie ck \texttt{ on } w'$, if and only if $w \bowtie w'$.*

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

4.3.1 Buffer Processes

When two processes communicate with synchronizable 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 $w \preceq w'$. A *buffer* $\texttt{buffer}_{w,w'}$ is a process with the clock type $\texttt{buffer}_{w,w'} : \forall\beta.\beta \texttt{ on } w \to \beta \texttt{ on } w'$ 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 $f : ck \to \alpha \texttt{ on } w$ and $f' : \alpha \texttt{ on } w' \to ck'$, with $w \bowtie w'$ and $w \preceq w'$. There exists a buffer $\texttt{buffer}_{w,w'} : \forall\beta.\beta \texttt{ on } w \to \beta \texttt{ on } w'$ such that $f' \circ \texttt{buffer}_{w,w'} \circ f$ 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 $(w_i)_{1 \le i \le n}$ and $(r_i)_{0 \le i \le n}$. 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' reads the data in x_i: $r_i[j] = 1$ means that the data in register x_i is read at cycle j. For a sequence of pushes and pops imposed by clocks w and w', the following case distinction simulates a FIFO on the x_i registers* statically *controlled through clocks w_i and r_i:*

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

`PUSH`: $w[j] = 1$ **and** $w'[j] = 0$**.** *Some data is written into the buffer and stored in register x_1, all the data in the buffer being pushed from x_i into x_{i+1}. Thus $x_i = x_{i-1}$ and $x_1 = input$, $\forall i > 2, w_i[j] = w_{i-1}[j-1]$, $w_1[j] = 1$ and $r_i[j] = 0$.*

`POP`: $w[j] = 0$ **and** $w'[j] = 1$**.** *Let $p = \max(\{0\} \cup \{1 \le i \le n | w_i[j-1] = 1\})$. 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 $r_0[j] = 1$, $w_i[j] = w_i[j-1]$. Conversely, if $p > 0$, $\forall i \neq p, r_i[j] = 0$, $r_p[j] = 1$, $\forall i \neq p, w_i[j] = w_i[j-1]$ and $w_p[j] = 0$. Registers x_i are left unchanged (notice this is not symmetric to the `PUSH` operation).*

`POP; PUSH`: $w[j] = 1$ **and** $w'[j] = 1$**.** *This case boils down to the implementation of a `POP` followed by a `PUSH`, as defined in the two previous 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*

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

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

We structurally extend this definition to stream clocks ck on w and ck on w' where $w <: w'$.[7]

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

$$\text{(SUB)}\quad \frac{P,H \vdash e : ck \text{ on } w \quad w <: w'}{P,H \vdash e : ck \text{ on } w'}$$

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 $(0-)$synchronous calculus since, assuming x has some clock α, α on (01) cannot be unified with α on $1(10)$.

$$\begin{array}{l}\texttt{let node } f(x) = y \texttt{ where}\\ y = (x \texttt{ when } (01)) + (x \texttt{ when } 1(10))\end{array}$$

Let e_1 denote expression (`x when` (01)) and e_2 denote expression (`x when` $1(10)$), and let us generate the type constraints for each construct in the program:

1. (NODE): suppose that the signature of f is of form $f : \alpha \to \alpha'$;
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 $1(10)$ for the clock of e_2;
4. (SUB): because (01) and $1(10)$ are synchronizable, the two clocks $ck_1 = \alpha$ on (01) and $ck_2 = \alpha$ on $1(10)$ can be resynchronized into α on (01), since $(01) <: (01)$ and $1(10) <: (01)$.

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

Considering the downscaler 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:

$$\text{(CTR)}\quad \frac{P,H \vdash e_1 : ck \text{ on } w_1 \quad P,H \vdash e_2 : ck \text{ on } w_2 \quad w_1 <: 0^d w_2}{P,H \vdash e_1 \texttt{ at } e_2 : ck \text{ on } 0^d w_2}$$

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

$$\begin{array}{l}\texttt{let node } f(x) = y \texttt{ at } (x \texttt{ when } (1001)) \texttt{ where}\\ y = (x \texttt{ when } (01)) + (x \texttt{ when } 1(10))\end{array}$$

We previously computed that $(x \texttt{ when } (01)) + (x \texttt{ when } 1(10))$ has signature $\alpha \to \alpha$ on (01), and (01) does not unify with (1001). Rule (CTR) yields

$$\frac{P,H \vdash y : a \text{ on } (01), x \texttt{ when } (1001) : a \text{ on } (1001) \quad (01) <: 0(1001)}{P,H \vdash y \texttt{ at } (x \texttt{ when } (1001)) : a \text{ on } 0(1001)}$$

Finally, $f : \forall\alpha.\alpha \to \alpha$ on $0(1001)$. Indeed, one cycle delay is the minimum to allow synchronization with the imposed output clock.

Relaxed Clock Calculus Rules The predicate $P,H \vdash_s e : ct$ 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 $\vdash_s$ judgments instead of $\vdash$.

$$\text{(SUB)}\quad \frac{P,H \vdash_s e : ck \text{ on } w_1 \quad w_1 <: w_2}{P,H \vdash_s e : ck \text{ on } w_2}$$

$$\text{(CTR)}\quad \frac{P,H \vdash_s e_1 : ck \text{ on } w_1 \quad P,H \vdash_s e_2 : ck \text{ on } w_2 \quad w_1 <: 0^d w_2}{P,H \vdash_s e_1 \texttt{ at } e_2 : ck \text{ on } 0^d w_2}$$

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 `fby` Operator Notice `fby` is considered a length preserving function in data-flow networks, hence its clock scheme $\forall\alpha.\alpha \times \alpha \to \alpha$ in the 0-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 $1(0)e \times \alpha \to \alpha$, $\forall\alpha.\alpha$ on $1(0) \times \alpha$ on $0(1) \to \alpha$, or $\forall\alpha.\alpha \times \alpha$ on $0(1) \to \alpha$. The first two signatures require the first argument to be present at the very first instant only, 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 α on $(1) <: \alpha$ on $0(1)$. This highlights the fact that the `fby` 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: $\text{delay}(w,w') = \min\{l \mid w \preceq 0^l w'\}$. When $w \preceq w'$, no delay is necessary. Note that in general, $\text{delay}(w,w') \neq \text{delay}(w',w)$.

[7] Yet this definition does not directly extends to stream clocks with different variables.

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*

$$delay(w, w') = \max(\max_p([w]_p - [w']_p), 0).$$

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 $w \preceq v$ since for all p, $[v]_p = d + [w']_p$. Moreover, d is minimal: suppose there exists p such that $d - 1 < [w]_p - [w']_p$, then $v' = 0^{d-1} w'$ satisfies $[v]_p = d - 1 + [w']_p < [w]_p$. Thus, $w \not\preceq v'$.* $\square$

For the simplified downscaler, the minimal delay to resynchronize the vertical filter with the output process is 0^{9603}, since 9603 (clock cycles) is the minimal value of d such that $0^{9600}(10000100000010000000100) \preceq 0^d(100000)$. 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 ($\texttt{on}$) when unifying stream clock types of the form $ck \texttt{ on } w$ and $ck' \texttt{ on } w'$. Two cases must be considered.

First of all, unifying $\alpha \texttt{ on } w$ and $\alpha \texttt{ on } w'$ returns true if and only if $w = w'$.

In the most general case, assume α and α' are clock variables (clocks can be normalised, thanks to the associativity of $\texttt{on}$). Equation $\alpha \texttt{ on } w = \alpha' \texttt{ on } w'$ *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 $w \preceq w'$, it is always possible to delay the p-th 1 in w (resp. w') until the p-th 1 in w' (resp. w) through the insertion of 0s in α (resp. in α'). Let us define the subsampling relation $\leq_{SS}$, such that

$$a \leq_{SS} a' \iff \exists \alpha, a = \alpha \textit{ on } a'.$$

Note that if $a \leq_{SS} a'$ then $a' \preceq a$, but the converse is not true: $(01) \preceq (0011)$ and there is no solution α such that $(0011) = \alpha \texttt{ on } (01)$.

PROPOSITION 9. *Relation $\leq_{SS}$ is a partial order.*

Proof. *$\leq_{SS}$ is trivially reflexive and transitive. Antisymmetry holds because $\preceq$ is a partial order, and $a \leq_{SS} a'$ implies $a' \preceq a$.* $\square$

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 ($\leq_{SS}$ is an upper semi-lattice), yet the expression of $m = v \texttt{ on } w = v' \texttt{ on } w'$ does not lead to a unique choice for m and for the maximal unifiers v and v'. 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 (v, v') is the chosen solution, the unification of $a \texttt{ on } w$ and $a' \texttt{ on } w'$ yields a unique clock type $\alpha \texttt{ on } v \texttt{ on } w = \alpha \texttt{ on } v' \texttt{ on } w'$, and every occurence of a (resp. a') is replaced by $\alpha \texttt{ on } v$ (resp. $\alpha \texttt{ on } v'$).

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 $m = v_1 \texttt{ on } w = v'_1 \texttt{ on } w' = v_2 \texttt{ on } w = v'_2 \texttt{ on } w'$; we have $v_1 \bowtie v_2$ and $v'_1 \bowtie v'_2$.*

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 $\mathcal{V}$ and $\mathcal{V}'$ through the following recurrent equations:

$$\begin{aligned}
\mathcal{V}(0^d 1.w, 0^d 0^{d'} 1.w') &= 1^d 0^{d'} 1.\mathcal{V}(w, w') \\
\mathcal{V}(0^d 0^{d'} 1.w, 0^d 1.w') &= 1^d 1^{d'} 1.\mathcal{V}(w, w') \\
\mathcal{V}'(0^d 1.w, 0^d 0^{d'} 1.w') &= 1^d 1^{d'} 1.\mathcal{V}'(w, w') \\
\mathcal{V}'(0^d 0^{d'} 1.w, 0^d 1.w') &= 1^d 0^{d'} 1.\mathcal{V}'(w, w')
\end{aligned}$$

Let $\mathcal{M}(w, w')$ denote the unifier

$$\mathcal{M}(w, w') = \mathcal{V}(w, w') \texttt{ on } w = \mathcal{V}'(w, w') \texttt{ on } w'.$$

The computation of $\mathcal{V}$ and $\mathcal{V}'$ terminates on periodic words because there are a finite number of configurations (bounded by the product of the period lenghts of w and w').

E.g., $a \texttt{ on } (1000) = a' \texttt{ on } 0(101)$:

w	1 0 0 0 1 0 0 0 1 0 0 0 0 ...	(1000)
w'	0 1 0 1 1 0 1 1 0 1 1 0 1 ...	0(101)
$\mathcal{V}(w, w')$	0 1 1 1 1 1 1 1 1 1 1 1 1 ...	0(1)
$\mathcal{V}'(w, w')$	1 1 1 0 0 1 0 0 0 1 1 0 0 ...	1(11001000)
$\mathcal{M}(w, w')$	0 1 0 0 0 1 0 0 0 1 0 0 0 ...	0(1000)

PROPOSITION 11. *For all w, w', p,*

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

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

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

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

Common subsamples of w and w' form a complete lower semi-lattice structure for $\preceq$, $\mathcal{M}(w, w')$ being the bottom element.

$\mathcal{M}$ is also associative: $\mathcal{M}(w, \mathcal{M}(w', w'')) = \mathcal{M}(\mathcal{M}(w, w'), w'')$. (Hence the complete lower semi-lattice structure for $\preceq$ holds for common subsamples of any finite set of infinite binary words.)

Proof. *We proceed by induction on the position of the p-th 1. Consider a infinite binary word $m' = u \texttt{ on } w = u' \texttt{ on } w'$. By construction of m, $[m]_1 = \max([w]_1, [w']_1)$, hence $[m]_1 \leq [m']_1$. Assume all common subsamples of w and w' are subsamples of m up to their p-th 1, and that $[m]_p \leq [m']_p$ for some $p \geq 1$. Proposition 11 tells that m is identical to either w or w' between its p-th and $p+1$-th 1; hence common subsamples of w and w' are subsamples of m up to the next 1; and since $w \preceq m'$ and $w' \preceq m'$ ($\leq_{SS}$ is a reversed sub-order of $\preceq$), we get $[m]_{p+1} \leq [m']_{p+1}$, hence $m \preceq m'$ by induction on p.*

Associativity derives directly from Proposition 11. $\square$

4.3.5 Resolution of the System of Clock Constraints

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

$$
\begin{array}{ll}
\text{(CYCLE)} & S+\{\alpha \text{ on } w_1 <: \alpha \text{ on } w_2\} \rightsquigarrow S \quad \text{if } w_1 <: w_2 \\
\text{(SUP)} & S+\{\alpha \text{ on } w_1 <: \alpha',\ \alpha \text{ on } w_2 <: \alpha'\} \rightsquigarrow S+\{\alpha \text{ on } w_1 \sqcup w_2 <: \alpha'\} \quad \text{if } w_1 \bowtie w_2 \\
\text{(INF)} & S+\{\alpha' <: \alpha \text{ on } w_1,\ \alpha' <: \alpha \text{ on } w_2\} \rightsquigarrow S+\{\alpha' <: \alpha \text{ on } w_1 \sqcap w_2\} \quad \text{if } w_1 \bowtie w_2 \\
\text{(EQUAL)} & S \rightsquigarrow S\left[\begin{array}{l}\alpha_1' \text{ on } v_1/\alpha_1 \\ \alpha_2' \text{ on } v_2/\alpha_2\end{array}\right] \quad \text{if } S = S'+I_1+I_2,\ \begin{array}{l} I_1 = \{\alpha_1 \text{ on } w_1 <: ck_1\} \text{ or } \{ck_1 <: \alpha_1 \text{ on } w_1\} \\ I_2 = \{\alpha_2 \text{ on } w_2 <: ck_2\} \text{ or } \{ck_2 <: \alpha_2 \text{ on } w_2\}\end{array},\ \begin{array}{l}\alpha_1 \neq \alpha_2 \\ w_1 \neq w_2\end{array},\ \begin{array}{l} v_1 = \mathcal{V}(w_1,w_2) \\ v_2 = \mathcal{V}'(w_1,w_2)\end{array} \\
\text{(CUT)} & S+\{\alpha_1 \text{ on } w <: \alpha_2 \text{ on } w\} \rightsquigarrow S+\{\alpha_1 <: \alpha_3 \text{ on } u_1,\ \alpha_3 \text{ on } u_2 <: \alpha_2\} \quad \text{if } \alpha_1 \neq \alpha_2,\ u_1 = \mathcal{U}_{\max}(w),\ u_2 = \mathcal{U}_{\min}(w) \\
\text{(FORK)} & S+\{\alpha <: \alpha_1 \text{ on } w,\ \alpha <: \alpha_2 \text{ on } w\} \rightsquigarrow S[\alpha_3 \text{ on } u \text{ on } w/\alpha]+\{\alpha_3 \text{ on } u <: \alpha_1,\ \alpha_3 \text{ on } u <: \alpha_2\} \quad \text{if } \alpha_1 \neq \alpha_2,\ u = \mathcal{U}_{\min}(w) \\
\text{(JOIN)} & S+\{\alpha_1 \text{ on } w <: \alpha,\ \alpha_2 \text{ on } w <: \alpha\} \rightsquigarrow S[\alpha_3 \text{ on } u \text{ on } w/\alpha]+\{\alpha_1 <: \alpha_3 \text{ on } u,\ \alpha_2 <: \alpha_3 \text{ on } u\} \quad \text{if } \alpha_1 \neq \alpha_2,\ u = \mathcal{U}_{\max}(w) \\
\text{(SUBST)} & S \oplus I \rightsquigarrow S[ck/\alpha] \quad \text{if } I = \{\alpha <: ck\} \text{ or } \{ck <: \alpha\},\ \alpha \notin FV(ck)
\end{array}
$$

Figure 8. Clock constraints resolution

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 $ck_i <: ck_i'$. 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:*

$$S \quad ::= \quad \{ck_1 <: ck_1', \ldots, ck_n <: ck_n'\}$$

We write $S+\{ck_1 <: ck_2\}$ for the extension of a system S with the inequation $\{ck_1 <: ck_2\}$. We write $S \oplus \{ck_1 <: ck_2\}$ for $S+\{ck_1 <: ck_2\}$ such that S does not contain a directed chain *of inequations from any free variable in ck_1 to any free variable in ck_2. For example, $S \oplus \{\alpha_1 <: \alpha_2 \text{ on } w_2\}$ means that, in S, α_1 never appear on the left of an inequation that leads transitively to an inequation where α_2 appears on the right.*

A system S is satisfiable if there exists a substitution ρ 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 iff $S[\alpha' \text{ on } w/\alpha]$ is satisfiable.*

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

Let us eventually define three functions useful to bound the set of subsamples of a given word: $\mathcal{U}_{\min}$, $\mathcal{U}_{\max}$ and Δ are defined recursively as follows:

$$
\begin{array}{lcl}
\mathcal{U}_{\min}(0^a1^b.w) & = & 1^a0^a0^b1^b.\mathcal{U}_{\min}(w) \\
\mathcal{U}_{\max}(0^a1^b.w) & = & 0^a0^b1^a1^b.\mathcal{U}_{\max}(w) \\
\Delta(0^c1^d.u, 0^a1^b.w, r) & = & 1^a1^b0^{c'}1^a1^b1^d.\Delta(u, w, r+f-\lfloor r+f\rfloor) \\
 & & \text{with } q = \frac{2c(a+b)}{d},\ c' = c+\lfloor r+q\rfloor \\
 & & \text{and } f = q-\lfloor q\rfloor
\end{array}
$$

Notice Δ — 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, $\mathcal{U}_{min}(w) \bowtie \mathcal{U}_{max}(w)$, $\mathcal{U}_{min}(w) <: \mathcal{U}_{max}(w)$, and $\mathcal{U}_{min}(w) \text{ on } w = \mathcal{U}_{max}(w) \text{ on } w$.*

For all u, w, $\Delta(u,w,0)$ is an infinite periodic binary word and is synchronizable with u.

Proof. *The first part of the proposition is proven inductively on the position of 1s in the subsampling.*

The second part is a consequence of the definition of c', designed to match the asymptotic rate of 1s in u (through the propagation of r, the fractional part of the asymptotic rate). □

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

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

$$\mathcal{U}_{min}(w) \text{ on } w <: v \text{ on } w \implies \mathcal{U}_{min}(w) <: v$$

and

$$v \text{ on } w <: \mathcal{U}_{max}(w) \text{ on } w \implies v <: \mathcal{U}_{max}(w).$$

For all u, v, w,

$$u \text{ on } \mathcal{U}_{min}(w) \text{ on } w <: v \text{ on } w \implies \Delta(u,w,0) \text{ on } \mathcal{U}_{min}(w) <: v$$

and

$$v \text{ on } w <: u \text{ on } \mathcal{U}_{max}(w) \text{ on } w \implies v <: \Delta(u,w,0) \text{ on } \mathcal{U}_{max}(w).$$

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

For the second pair of implications, observe that

$$1^a1^b0^{c'}1^a1^b \text{ on } 1^a0^a0^b1^b.U = 1^a0^b0^{c'}0^a1^b$$

and

$$1^a1^b0^{c'}1^a1^b \text{ on } 0^a0^b1^a1^b.U = 0^a0^b0^{c'}1^a1^b,$$

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

$$v' \text{ on } w = \Delta(u,w,0) \text{ on } \mathcal{U}_{min}(w) \text{ on } w = \Delta(u,w,0) \text{ on } \mathcal{U}_{max}(w) \text{ on } w.$$

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

Let us finally define the simplification relation $\rightsquigarrow$ 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 $S \rightsquigarrow S'$ 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). *Presevation 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 $\mathcal{U}_{min}$ and $\mathcal{U}_{max}$, the right-hand side of the relation is a sufficient condition of satisfiability.*

Conversely, consider a solution $\alpha_1 = \alpha \text{ on } v_1$ and $\alpha_2 = \alpha \text{ on } v_2$. Let $V_1 = \mathcal{V}(v_1, \mathcal{U}_{min}(w))$ and $V_1' = \mathcal{V}'(v_1, \mathcal{U}_{min}(w))$, and replace α by $\alpha' \text{ on } V_1$. We have $\alpha_1 = \alpha' \text{ on } V_1' \text{ on } \mathcal{U}_{min}(w)$. Let us choose $\alpha_3 = \alpha' \text{ on } \Delta(V_1', w, 0)$; From Proposition 14, we have $\alpha_1 <: \alpha' \text{ on } V_1' \text{ on } \mathcal{U}_{max}(w) <: \alpha_3 \text{ on } \mathcal{U}_{max}(w)$.

We also have $V_1' \text{ on } \mathcal{U}_{min}(w) \text{ on } w <: V_1 \text{ on } v_2 \text{ on } w$, hence Proposition 14 yields $\Delta(V_1', w, 0) \text{ on } \mathcal{U}_{min}(w) <: V_1 \text{ on } v_2$. Since $\alpha_2 = \alpha' \text{ on } V_1 \text{ on } v_2$, we have $\alpha_3 \text{ on } \mathcal{U}_{min}(w) <: \alpha_2$. The right-hand side of the relation is thus satisfiable.

(FORK) and (JOIN). *The proof is very similar: choosing the same α_3 satisfies both inequalities on α_1 and α_2 simultaneously.*

(SUBST). *Consider the form of the inequality I on α. 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.

□

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 $S \rightsquigarrow S'$ 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 $P, H \vdash_s e : ct$.*

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 $ck \text{ on } w$ and the output period $ck \text{ on } w'$, with $w \preceq w'$. To fully synchronize the communication, we insert a new buffer node $\texttt{buffer}_{w,w'}$ with clock $\forall\beta.\beta \text{ on } w \rightarrow \beta \text{ on } w'$; w (resp. w') states when a *push* (resp. *pop*) occurs.

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

$$size(w, w') = \max(\max_{p,q}(\{q - p \mid [w']_p \geq [w]_q\}), 0).$$

Communication from w to w′ is called $size(w, w')$-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 $0^{9600}(1000010000000\ 10000000100)$ may take at most one advance tick with respect to clock $0^{9603}(100000)$. 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 $P, H \vdash_s e : ct \Rightarrow e'$, 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:

$$\text{(TRANSLATION)} \quad \frac{P, H \vdash_s e : ck \text{ on } w \Rightarrow e' \quad w <: w'}{P, H \vdash_s e : ck \text{ on } w' \Rightarrow \texttt{buffer}_{w,w'}(e')}$$

Other rules are simple morphisms.[8]

[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 (TRANSLATION) rule indirectly.

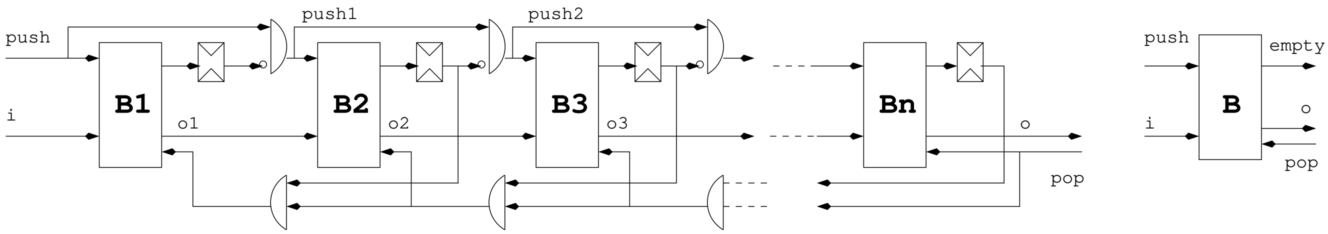

Figure 9. A synchronous buffer

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 of $(0-)$synchronous communication between periodic clocks (with statically known periodic clocks). We propose an alternative construction where the presence or absence of data is captured by dynamically computed clocks. The memory and code size become 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.

```
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 10. Synchronous buffer implementation

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 explicitely 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.

[9] LUCID SYNCHRONE [13]; distribution and reference manual available at `www.lri.fr/~pouzet/lucid-synchrone`.

5.3 Correctness

We define judgment $P,H \vdash e : ct$ to denote that expression e has clock ct 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 $P,H \vdash_s e : ct \Rightarrow e'$ then $P,H \vdash e' : ct$.*

The proof derives from the subtyping rule underlying $\vdash_s$ 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 sudies, 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 correct-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 SYNCHRONE 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

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[3] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[4] G. Berry. *The Foundations of Esterel*. MIT Press, 2000.

[5] J.-Y. Brunel, W. M. Kruijtzer, H. J. H. N. Kenter, F. Pétrot, L. Pasquier, E. A. de Kock, and W. J. M. Smits. COSY communication IP's. In *37th Design Automation Conference (DAC'00)*, pages 406–409, Los Angeles, California, June 2000.

[6] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):155–182, 1994.

[7] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *EMSOFT'01*, volume 2211 of *LNCS*, Lake Tahoe, October 2001. Springer-Verlag.

[8] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 226–238. ACM Press, 1996.

[9] Z.S. Chamski, M. Duranton, A. Cohen, C. Eisenbeis, P. Feautrier, and D. Genius. Application-domain-driven system design for pervasive video processing. *Ambient intelligence: impact on embedded system design*, pages 251–270, 2003.

[10] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[11] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. Synchronization of periodic clocks. In *ACM Conf. on Embedded Software (EMSOFT'05)*, Jersey City, New York, September 2005.

[12] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *EMSOFT'04*, Pisa, Italy, september 2004.

[13] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In Rajeev Alur and Insup Lee, editors, *EMSOFT'03*, volume 2855 of *Lecture Notes in Computer Science*, pages 134–155. Springer, 2003.

[14] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. Yapi: Application modeling for signal processing systems. In *37th Design Automation Conference*, Los Angeles, CA, june 2000. ACM Press.

[15] D. G. Messerschmitt E. A. Lee. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–25, 1987.

[16] K. Goossens, G. Prakash, J. Röver, and A. P. Niranjan. Interconnect and memory organization in SOCs for advanced set-top boxes and TV — evolution, analysis, and trends. In Jari Nurmi, Hannu Tenhunen, Jouni Isoaho, and Axel Jantsch, editors, *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 15, pages 399–423. Kluwer, April 2004.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[18] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[19] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design*, April 2003.

[20] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1), 1991.

[21] A.J.M. Moonen, M. Bekooij, and J. van Meerbergen. Timing analysis model for network based multiprocessor systems. In *proceedings of ProRISC, 15th annual Workshop of Circuits, System and Signal Processing*, pages pages 91 – 99, Veldhoven, The Netherlands, November 2004. ISBN: 90-73461-43-X.

[22] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[23] F. Pottier. Simplifying subtyping constraints. In *ACM Intl. Conf. on Functional Programming (ICFP'96)*, volume 31(6), pages 122–133, 1996.

[24] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[25] J. E. Vuillemin. On circuits and numbers. *IEEE Trans. Comput.*, 43(8):868–879, 1994.