HAL open science

Synchronizing Periodic Clocks in Kahn Networks

Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, Marc Pouzet

To cite this version:

Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, et al.. Synchronizing Periodic Clocks in Kahn Networks. [Research Report] RR-5603, INRIA. 2005, pp.38. inria-00070404

HAL Id: inria-00070404

https://inria.hal.science/inria-00070404v1

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Synchronizing Periodic Clocks in Kahn Networks

Albert Cohen, Marc Duranton[1], Christine Eisenbeis, Claire Pagetti, Florence Plateau[2] and Marc Pouzet[2]

N° 5603

2005

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5603--FR+ENG

Synchronizing Periodic Clocks in Kahn Networks

Albert Cohen, Marc Duranton[1], Christine Eisenbeis, Claire Pagetti, Florence Plateau[2] and Marc Pouzet[2]

Thème COM — Systèmes communicants
Projet Alchemy
[1] Philips Research Laboratories, Eindhoven, The Netherlands
[2] LIP6, Pierre and Marie Curie University, Paris, France

Rapport de recherche n° 5603 — 2005 — 38 pages

Abstract: We propose a programming model and language dedicated to high-performance streaming applications. In particular, we study real-time video-streaming for embedded media devices, including high-definition TVs. This language builds on the synchronous programming model and on domain-specific knowledge — periodic evolution of streams — to allow correct-by-construction properties of the application to be proven by the compiler. These properties include buffer requirements and delays between input and output streams. Correctness of the implementation is difficult to assess with traditional (asynchronous) approaches.

Such properties are tedious to analyze by hand, due to the combinatorics of video filters, multiple data rates and formats. For example, the design of communicating buffers between filtering processes (image scaling, quality enhancement, etc.) whose clocks do not strictly match is tedious and error-prone. Two communicating periodic processes are defined as n-*synchronous* if they can be implemented in the ordinary (0-)synchronous model with a FIFO buffer of size n. 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. This relaxation is associated with a subtyping rule in the clock calculus. Delay, buffer insertion and control code for these buffers are automatically inferred from the clock types through a systematic program transformation.

Key-words: Kahn Networks, Synchronous Programming Languages, Real Time, High-Performance Embedded Systems, Video Streaming

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,

Synchronisation des horloges périodiques dans les processus de Kahn

Résumé :

Nous présentons un modèle de programmation et un langage dédié aux applications de traitement de flux. En particulier, nous étudions les programmes vidéo destinés aux systèmes enfouis multimédia, par exemple pour la télévision haute définition. Ce langage se base sur le modèle de programmation synchrone et sur la spécificité du domaine - les flux ont un cadencement périodique; il permet de concevoir des programmes corrects par construction, dont les propriétés de synchronicité sont prouvées par le compilateur. Ces propriétés sont obtenues par insertion de buffers et de retards entre les flux sortants et flux entrants. Elles sont très difficiles à atteindre dans les approches (asynchrones) traditionnelles.

L'analyse à la main de ces propriétés est très délicate à cause de la combinatoire induite par le cadencement des filtres, et les multiples débits et formats des données. En particulier, l'insertion des buffers et leur programmation entre les processus filtrants (réduction/agrandissement d'images, amélioration de la qualité des images) est fastidieuse et facilement source d'erreurs lorsque les cadencements du flux entre ces processus ne sont pas strictement en phase. Deux processus communiquants sont dits n-synchrones si ils peuvent être implémentés dans le modèle ordinaire (0-synchrone) en insérant un buffer de taille n. Nous enrichissons un langage à flots de données, synchrone en lui ajoutant la notion d'horloges périodiques, puis construisons un calcul d'horloge (système de types pour les horloges) qui permet la composition de processus non strictement synchrones. Cette relaxation se ramène à une simple règle de sous-typage dans le calcul d'horloges. L'insertion de retards, de buffers, ainsi que le code de contrôle de ces buffers sont automatiquement inferés à partir des types des horloges par une transformation de programmes systématique.

Mots-clés : Réseaux de processus de Kahn, Langages de programmation synchrones, temps réel, systèmes embarqués à hautes performances, traitement de flux vidéo.

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 be common in consumer devices in the midterm. In this signal-processing domain, compute-intensive kernels used to be mapped to specific fixed hardware (ASIC) for performance, cost, power and predictability reasons (real-time). Yet the combined increase in mask costs and in the variability of supported algorithms leads to a strong pressure towards programmable, domain-specific designs, balancing the software and hardware shares.

Unfortunately, general-purpose architectures and compilers are not suitable for the design of real-time *and* high-performance (massively parallel) *programmable* system-on-chip [10]. To achieve tera-operations per second, the mid-term multi-core VLIW or superscalar architectures will require both a high frequency and a large die area, with a power budget incompatible with most embedded markets. Achieving a higher compute density and still preserving programmability is a challenge for the choice of an appropriate architecture, programming language and compiler. Typically, a low power design requires the clock frequency of the chip to be as low as possible, in the same order of magnitude as the pixel clock of the video application; this means that thousands of operations per cycle must be sustained in real-time, exploiting multiple levels of parallelism in the compute kernel.

Interestingly, the synchronous execution paradigm [3] 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 [4], LUSTRE (Scade) [17] and ESTEREL [5]. It is thus natural to investigate the applicability of such languages for the design of warrantable high-performance systems like high-end video applications. This paper is focused on model, language and compilation issues. Optimization, automatic parallelization and mapping are left for future work.

To enforce real-time and resources properties, the synchronous paradigm assumes a common clock for all registers, and an overall predictable hardware where communications and computations can be proven to take less than a 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 [11]. This fact has a strong impact on the formalization of synchronous execution itself and on the associated compilation (code generation) strategies [19].

Due to the complexity of high-performance applications and to the intrinsic combinatorics of synchronous execution, we will show that *multiple clock domains* have to be considered at the *application level*. 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). We thus need 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 requires 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 simple flow-preservation constraint, through the automatic inference of bounded delays and FIFO buffers. More technically, our main contributions are the following.

- 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.[1] This principle is currently restricted to periodic clocks defined as periodic infinite binary words.
- We define a relaxed synchronous functional programming language, the clock calculus of which is extended with n-synchrony. In practice, the classical synchronous clock calculus can be defined as a type system. In order to allow n-synchrony, we extend it with two additional subtyping rules. The resolution is then performed using an ad hoc unification.
- We show that every n-synchronous program can be transformed into a synchronous one (0-synchronous), replacing bounded buffers by some synchronous code.
  - The first one is mostly theoretical and reduces the n-synchronous program to a synchronous one (0-synchronous) with *statically known periodic clocks*, expressible in the original core calculus. Unfortunately, this strategy requires an exponential memory and code size to characterize all control states of multiple communicating FIFO buffers.
  - The second strategy reduces the program to a synchronous one expressible in an classical (non periodic) synchronous calculus, but implements FIFO buffers where the presence or absence of data is captured by dynamically computed clocks. The memory and code size become linear in the total buffer size and appropriate for a practical implementation. Without our language extensions to reason about n-synchronous periodic clocks, it is hard to prove that the code actually behaves as a loss-less FIFO when at most n tokens are sent and not yet received (using abstract interpretation or model-checking tools).

The structure of the document is the following. In section 2, we motivate the use of the model of n-synchrony for programming high-performance video applications through the presentation of a downscaler application. Section 3 introduces a core synchronous language *a la* LUSTRE and its semantics, then presents an associated calculus on periodic

[1] 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.

clocks. Section 4 is our main contribution: it extends this calculus to combine streams with n-synchronizable clocks. Section 5 describes code generation strategies (for any n-synchronizable program) based on translation to a purely synchronous, by automatically inserting buffers with minimal size. Related works are discussed in Section 6, before we conclude in Section 7.

2 Motivation

Our main motivating applications are video stream processing for high-definition TV [16]. These 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 for progressive screen such as flat panel displays, sharpness improvement, color enhancement, etc.). Processing requires considerable resources, and involves a variety of pipelined algorithms on multidimensional streams.

Philips's line of home entertainment engines is built on media processors for high-performance and high-quality multimedia applications. For instance, the Nexperia Home family of chips (such as the PNX8550) [25] can be found on TV-centric products such as analog, analog/digital, and digital TV sets and receivers, connected digital TVs, home multimedia servers, and home entertainment centers.

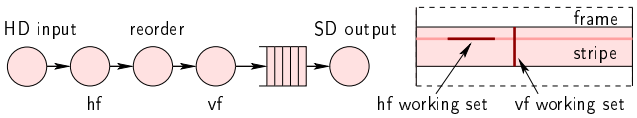


Figure 1: The downscaler

These applications involve a set of scalers resize images in real-time. Our running example is a classical downscaler [10], depicted in Figure 1. It converts a high definition (HD) video signal, 1920×1080 pixels per frame, into a standard definition (SD) output for TV screen, that is 720×480:[2]

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 downto 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 satisfied, and to optimize

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

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 three 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, allowing to bound resource requirements and assess real-time execution. Yet, we will 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 [10]):

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

Figure 2 shows a first version of the horizontal filter, implemented in a strictly synchronous language such as Lucid Synchrone [9] or Lustre [17]: at every top, a new pixel `p` is stored in the first position o_1 of an internal memory of size 6; the output `o` is equal to the interpolation of six consecutive pixels. We do not detail the access to the coefficient which can be encoded in the synchronous function f. The implementation of f is out of the scope of this paper; we will assume it sums its 6 arguments. An execution is defined by the stream of values for each variable, see Figure 2.

In the synchronous model, each variable/expression is characterized both by its stream of values and by its *clock*, relative to the global clock, called `base`. The clock of any expression e is an infinite stream of 0 and 1 where 0 stands for the absence and 1 for the presence. E.g., if `x` is an integer 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 often relative to some clock variables. E.g., the clock signature of the incrementation process defined by `(node inc x = x+1)` is $\forall\alpha.\alpha \rightarrow \alpha$.

Clearly, the horizontal filter `hf` must have a different clock signature, to match the production of 3 pixels for 8 input pixels. Moreover, the signal processing algorithm defines

```
node hf p = o where o1 = p
  and o2 = 0 fby o1
  and o3 = 0 fby o2
  and o4 = 0 fby o3
  and o5 = 0 fby o4
  and o6 = 0 fby o5
  and o = f(o1, o2, o3, o4, o5, o6)
```

p	3	4	7	5	6	10
o_1	3	4	7	5	6	10
o_2	0	3	4	7	5	6
o_3	0	0	3	4	7	5
o_4	0	0	0	3	4	7
o_5	0	0	0	0	3	4
o_6	0	0	0	0	0	3
o	3	7	14	22	28	38

Figure 2: First implementation of hf

```
node hf p = o where cnt = 1 fby
  if (cnt=8) then 1 else cnt+1
  (...)
  and o = f(o1, o2, o3, o4, o5, o6)
  when ((count=1) or
    (count=3) or (count=6))
```

p	3	4	7	5	6	10
o	3		7			14

Figure 3: Synchronous implementation and execution of hf

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 (for this particular scaling factor, in general the scaling factor is a parameter of the filter function). A solution should be to define an auxiliary stream `cnt` which counts the number inputs and guards the effective output of `o`; see Figure 3.

In practice, the considered applications 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 $30 \times 1920 \times 1080 = 62,208,000Hz$ — called the HD pixel clock — and SD pixels at $30 \times 720 \times 480 = 10,368,000Hz$ — called the SD pixel clock — i.e., 6 times slower.

Considering the downscaler example, 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 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. 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 the real pixel rate, including blanking periods [16].

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

3 Periodic Synchronous Clocks

This section introduces a simple data-flow functional language on infinite data streams. This language has a (strictly) synchronous semantics, enforced by a so-called *clock calculus*, a type system to reject non synchronous programs, following [9, 12].

3.1 Definitions and Notations

We consider *infinite binary words*, i.e., words of $(0+1)^\omega$. We are mostly interested in a subset of the binary words, called *infinite periodic binary words,* defined by the following grammar:

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

where $(v) = \lim_n v^n$ is the infinite repetition of *period* v, and u is a prefix of w. Note that words in $(0+1)^\omega$ correspond exactly with the diadic numbers whose interest for synchrony was first discovered by [27]. We denote by $\mathbb{Q}_2$ the set of infinite periodic binary words; this set coincide with the rational diadic numbers [27].

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. In the remaining we will always consider infinite periodic binary words with infinite number of 1's or equivalently, with period (v) containing at least one 1. This corresponds to removing the integer numbers from $\mathbb{Q}_2$ and considering only $\mathbb{Q}_2 - \mathbb{N}$.

There is 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 for any infinite periodic binary word w: it is the (unique) shortest representation of the form $u(v)$, which is equivalent to the representation with the shortest prefix and period.

We denote by $\sqsubseteq$ the prefix relation. Thus let w and w' be two binary words, $w \sqsubseteq w' \Rightarrow \exists v$ binary word (possibly infinite), such that $w.v = w'$. For instance, 010101 is a prefix of (01).

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 $\preceq$ defined 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 periodic binary words with

$$\forall p \geq 1, [w \sqcup w']_p = \max([w]_p, [w']_p)$$
$$\forall p \geq 1, [w \sqcap w']_p = \min([w]_p, [w']_p).$$

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

The set of infinite binary words is a complete lattice. We recall some general definitions on ordered sets.

Definition 1 (Lattice [14]) *1. A* poset $(L, \preceq)$ *is a set equipped with an order relation.*

2. A poset is a lattice $(L, \preceq, \sqcup, \sqcap)$ *iff every finite subset $S \subseteq L$ has a least upper bound $\sqcup S$ and a greatest lower bound $\sqcap S$.*

3. A lattice is complete $(L, \preceq, \bot, \top, \sqcup, \sqcap)$ iff every (possibly infinite) subset $S \subseteq L$ has a least upper bound $\sqcup S$ and a greatest lower bound $\sqcap S$. Hence $\bot = \emptyset$ is the infimum and $\top = \sqcup L$ is the supremum.

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

Indeed, by induction on the definition of the supremum and infimum, the upper and lower bounds are always defined for every finite subsets. Moreover $\top = (0)$ and $\bot = (1)$ since $[(0)]_p = \infty$ for all $p > 0$. For an infinite set S, $\mathit{inf}(S) = w$ where $[w]_p = \mathit{sup}\{[w']_p \mid w' \in S\}$ for every $p \in \mathbb{N}$.

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

Remark 1 *Any pair of infinite periodic binary words can be represented with prefixes and periods whose lengths satisfy some condition. Consider two infinite periodic binary words $w = u(v)$ and $w' = u'(v')$. We can transform the expressions of w and w' into particular representatives $a(b)$ and $a'(b')$ which ease some computations.*

We can choose for instance a, a', b, and b' such that $|a| = |a'| = max(|u|, |u'|)$ and $|b| = |b'| = lcm(|v|, |v'|)$ where lcm stands for least common multiple. *Indeed, suppose $|u| \leq |u'|$ (the other case is straightforward), $p = |u'| - |u|$ and $n = lcm(|v|, |v'|)$, then $w = u.v[1] \cdots v[p].\big((v[p+1] \cdots v[p+|v|])^{n/|v|}\big)$ and $w' = u'.(v'^{n/|v'|})$. E.g.,* 010(001100) *and* 10001(10) *become* 01000(110000) *and* 10001(101010).

Likewise, one may enforce that prefixes and suffixes have 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 $|u|_1 \leq |u'|_1$ and $|v|_1 \leq |v'|_1$, $p = |u'|_1 - |u|_1$, $r = [v]_p$, and $n = lcm(|v|_1, |v'|_1)$,

then $w = u.v[1]\cdots v[r].\big((v[r+1]\cdots v[r+|v|])^{n/|v|_1}\big)$ *and* $w' = u'.(v'^{n/|v'|_1})$. *E.g.,* 010(001100) *and* 10001(10) *become* 010001(100001) *and* 10001(101010).

The last interesting case is $w = a(b)$ *and* $w' = a'(b')$ *with* $|a|_1 = |a'|$ *and* $|b|_1 = |b'|$. *Indeed, suppose* $|u|_1 \leq |u'|$ *and* $|v|_1 \leq |v'|$, $p = |u'|_1 - |u|$, $r = [v]_p$, *and* $n = lcm(|v|_1, |v'|)$, *then* $w = u.v[1]\cdots v[r].\big((v[r+1]\cdots v[r+|v|])^{n/|v|_1}\big)$ *and* $w' = u'.(v'^{n/|v'|})$. *E.g.,* 010(001100) *and* 10001(10) *become* 0100011000011(000011) *and* 10001(10).

For the sake of simplicity, we will assume thereafter that every infinite periodic binary word has an infinite number of 1s, i.e., is period holds at least one 1.

3.2 Clocks and Periodic Clocks

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

$$\begin{array}{l} clk ::= w \mid clk\ on\ w \\ w \in (0+1)^\omega \end{array}$$

If ck is a clock and w is an infinite binary word, then ck `on` w denotes a *sub-sampled clock* of ck, where w is itself set on clock ck.

E.g., (01) `on` (101) = (010101) `on` (101) = (010001).

clk	0	1	0	1	0	1	0	1	0	1	...	(01)
w		1		0		1		0		1	...	(101)
clk `on` w	0	1	0	0	0	1	0	0	0	1	...	(0100)

Formally, *on* is inductively defined as follows:

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

Proposition 1 *Given* w *and* w' *two infinite binary words,* w `on` w' *is also an infinite binary word, satisfying the equation* $[w \texttt{ on } w']_p = [w']_{[w]_p}$ *for all* $p \geq 1$.

Indeed, if the n-th value of w is 0, the n-th value of w `on` w' is necessarily 0, otherwise it is the p-th 1 of w (for some $p \geq 1$). Since w' is covered at the rate of 1s in w, the n-th value of w `on` w' is the $[w]_p$-th 1 in w'.

The following result is a corollary of the previous property.

Proposition 2 (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 $[w_1]_{[w_2 \texttt{ on } w_3]_p} = [w_1]_{[w_2]_{[w_3]_p}} = [w_1 \texttt{ on } w_2]_{[w_3]_p}$.

However, the *on* operator is not commutative. We have the following result:

Lemma 1 *Let w, w' be two infinite binary words. We have $w \preceq w'$ if, and only if there exists $u \in (0+1)^\omega$ such that $w' = u$ on w.*

Proof. *The word $u = \prod_p 0^{[w']_p - [w]_p}.1$ answers the lemma.* □

A periodic clock is a clock whose stream is periodic. The periodic clocks are defined as follows:

$$clk ::= w \mid clk \texttt{ on } w$$
$$w \in \mathbb{Q}_2$$

The previous results on clocks hold and moreover Proposition 1 becomes an algorithm: 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 $|u_3| = |u_1|$, $|u_3|_1 = |u_2|_1$, $[u_3]_p = [u_2]_{[u_1]_p}$ and $|v_3| = |v_1|$, $|v_3|_1 = |v_2|_1$, $[v_3]_p = [v_2]_{[v_1]_p}$.

3.3 A Synchronous Data-Flow Kernel

We introduce a core data-flow language on infinite streams. Its syntax derives from [13]. Expressions (e) are made of constant streams (i), variables (x), pairs (e, e), local definitions of functions or stream variables (e `where` $x = e$),[3] applications ($e(e)$), initialized delays (e `fby` e) and the following sampling functions: e `when` pe is the sampled stream of e on some periodic clock 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 if e on the negation of periodic clock pe.

A program is made of a sequence of declarations of stream functions (`node` $x(x) = e$) and periodic clocks (`period` $p = pe$). For example `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` $half_2 = half$ `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.

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

We can easily program the downscaler in this language. It is much simpler than in a classical synchronous language, avoiding awkward auxiliary variables, see the horizontal filter in Figure 4.

[3]Corresponds to `let` x `=` e `in` e in OCaml [24].

```
node hf p = o where o1 = p
  (...)
  and o = f(o1,o2,o3,o4,o5,o6)
  when (10100100)
```

Figure 4: Periodic clock

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

Figure 5: Imposing clocks

This implementation matches the average 3 outputs (there are three 1s in the clock period) for 8 inputs (length of the period) of the horizontal filter, see also Figure 3.

To implement the rest of the downscaler in Figure 5, we use the `at` syntax to impose clocks to input and output streams. Node `main` states that the input is sampled at the fastest clock `base` = (1), whereas output pixels have to be produced at clock (100000), 6 times slower.

3.4 Clock Calculus

The denotational and synchronous (operational) semantics of our core data-flow language are built on classical theory of synchronous languages [13]. 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 synchronous circuits (i.e., expressions from e) [27].

This kernel is statically typed. Its typing rules are straightforward and are not given in this report. See [13] for example. In the same way, we do not consider causality and initialization problems nor the rejection of recursive stream functions. These classical analysis apply directly to our core language and they are orthogonal to synchrony.

The compilation process takes two steps.

1. Our *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.

3.4.1 Type System

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[4] are decomposed into clock schemes (σ) quantified over a set of clock variables (α) and unquantified clock types (ct). A clock may be a functional clock ($ct \rightarrow ct$), a product ($ct \times ct$) or a stream clock (ck). A stream clock may be the base clock (`base`), a sampled clock (ck `on` pe) or a clock variable (α).

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

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

Note that `base` can be considered as a shortcut for the period (1), that is, the boolean sequence made of true values. `base` plays a particular role in a system since it is finally set to a real-time clock such that nothing should run faster. This property is ensured by the structure of ck: in the main program, all the stream clocks will be of the form `base on` pe_1... `on` pe_n that is, sub-clocks of the `base` one. Moreover, such a clock expression simplifies to a clock expression of the form `base on` pe (applying property 2).

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 $\forall \alpha.\alpha \rightarrow \alpha$ `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 instantiated in several ways, replacing α by more precise stream clock type (e.g., the base clock or 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 (α) 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 by clock type expressions. $fgen(ct)$ returns a fully generalized clock type where every variable from ct is quantified universally.

When defining periods, we must take care that identifiers in periods 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 6 and are discussed below.

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

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

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

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

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

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

$$\text{(WHERE)}\ \frac{P, H, x : ct \vdash e : ct \quad P,\ H, x : ct \vdash e' : ct'}{P, H \vdash e'\ \texttt{where}\ x = e : ct'}$$

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

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

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

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

$$\text{(DEF)}\ \frac{H \vdash d_1 : H_1 \quad H, H_1 \vdash d_2 : H_2}{H \vdash d_1; d_2 : H_1, H_2} \qquad \text{(DEF')}\ \frac{P \vdash dp_1 : P_1 \quad P, P_1 \vdash dp_2 : P_2}{P \vdash dp_1; dp_2 : P_1, P_2}$$

$$\text{(CONSTRAINT)}\ \frac{H \vdash e : ck}{H \vdash e'\ \texttt{at}\ e : ck}$$

Figure 6: The 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))
- The rule (FBY) states the clock of e_1 `fby` e_2 is the one of e_1 and e_2. The 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 period. 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.
- The rule (APP) is the classical typing rule of ML type systems.
- The rule (WHERE) is the rule for recursive definitions.
- The rules (PAIR), (FST) and (SND) are the rules for pairs.
- Node declarations (rule (NODE)) are clocked as regular function definitions. We write $H, x : ct_1$ as the clock environment H extended with the association $x : ct_1$. Because node definitions only apply at top-level (and cannot be nested), we can generalize every variable appearing in the clock type [5].
- The following rules check that periods are well formed, that is, names in expressions of periods are first defined before being used.
- The rule (CONSTRAINT) for the syntax e_1 `at` e_2 states that the clock associated to e_1 is imposed by the clock of e_2.

Let us illustrate these definitions on the downscaler.

1. The clock of the input process (input) is the binary word the base clock `base` $= (1)$.
2. The horizontal filter has the following signature, corresponding to the effective synchronous implementation of the process: $\alpha \rightarrow \alpha$ `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 buffer operator the clock signature $\beta \rightarrow 0^{3600}\beta$.
4. The vertical filter produces 4 pixels from 9 pixels repeatedly across the 720 pixels of a stripe (6 lines). Its signature (corresponding to the synchronous implementation of the process) is:

$$\gamma \rightarrow \gamma \texttt{ on } (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720})$$

[5] 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.

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: $\gamma \rightarrow \gamma$ `on` (101001001).

5. Finally, the designer has required that if the global input is on clock ck, then the clock of the `output` process should be ck on (100000) — the 6 times sub-sampled input clock — tolerating an additional delay that must automatically be deduced from the clock calculus.

The composition of all 5 processes yield the following type constraints: `base` $= \alpha$, $\beta = \alpha$ `on` (10100100), $\gamma = \beta$ `on` $0^{3600}(1)$, and γ `on` (101001001) $=$ (100000). The next section addresses the resolution of these equations.

3.4.2 Unification

Unification in classical clock calculi is syntactical [12]. In our case, a syntactic unification of clock types 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 (on) when unifying clock stream types of the form ck `on` w and ck' `on` w'.

Four cases must be considered. First of all, unifying `base` `on` w and `base` `on` w' returns true if and only if $w = w'$. This is the same when unifying a `on` w and a' `on` w'.

Third, unifying a `on` w with `base` `on` w' yields three possible results: if w' is not a sub-sample of w, there is no solution; otherwise there may be one or more solutions, a particular one is selected as in the general case below.

In the most general case, assume a and a' are clock variables. Equation a `on` $w = a'$ `on` w' always have an infinite number of solutions (a, a'), and these solutions generate an infinite number of different infinite periodic binary words. Intuitively, a periodic sampling of w consists in the insertion of 0s in w, in a periodic manner. 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 a (resp. in a'). Let us define the sub-sampling relation $\leq_{SS}$, such that

$$a \leq_{SS} b \iff \exists \alpha, a = \alpha \; on \; b.$$

Note that if $a \leq_{SS} b$ then $a \preceq b$ and it is just an implication. E.g., (011) $\preceq$ (01) and there is no solution α such that (011) $= \alpha$ `on` (01). Thus because of lemma 1 the sub-sampling relation does not coincide with the precedence relation.

Proposition 3 *The relation $\leq_{SS}$ is a partial order.*

Indeed, it is trivially reflexive and transitive. For the antisymmetry, we need to prove that $x \leq_{SS} y$ and $y \leq_{SS} x$ implies $x = y$. Since, $x \leq_{SS} y$ implies $x \preceq y$ and $y \leq_{SS} x$ implies $y \preceq x$, and since $\preceq$ is a partial order, then $x = y$.

In a typical unification scheme, one would like to replace the above type equation by "the most general clock stream type satisfying the equation". This is unfortunately not possible in general, since $\leq_{SS}$ is not an upper semi-lattice.[6] Notice there is such a "most general clock" in the special case of $w = (1)$: all sub-samples of a' `on` w' are sub-samples of clock stream type a `on` $(1) = a$.

We 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 `on` w and a' `on` w' yields a unique clock stream type α `on` v `on` $w = \alpha$ `on` v' `on` w', and every occurence of a (resp. a') is replaced by α `on` v (resp. α `on` v' in the pair of clock types involved in the unification.

Yet we do not make an arbitrary choice for (v, v'). Intuitively, we select a periodic one that minimizes delay insertion in a `on` $w = a'$ `on` w'. This choice will be further motivated in Section 5.3. Formally, we inductively define a "greedy" unifier $(V(w, w'), V'(w, w'))$ as follows:

$$\begin{array}{rcl@{\qquad}rcl} V(1.w, 1.w') & = & 1.V(w, w') & V(0.w, 0.w') & = & 1.V(w, w') \\ V'(1.w, 1.w') & = & 1.V'(w, w') & V'(0.w, 0.w') & = & 1.V'(w, w') \\ V(1.w, 0.w') & = & 0.V(1.w, w') & V(0.w, 1.w') & = & 1.V(w, 1.w') \\ V'(1.w, 0.w') & = & 1.V'(1.w, w') & V'(0.w, 1.w') & = & 0.V'(w, 1.w') \end{array}$$

E.g., (1) `on` $(01) = 011(1110)$ `on` $10(101)$.

w	0	1	0	1	0	1	0	1	0	1	...	(01)
w'		1	0	1	0	1		1	0	1	...	10(101)
v	1	1	1	1	1	1	1	1	1	1	...	(1)
v'	0	1	1	1	1	1	0	1	1	1	...	011(1110)
v `on` w	0	1	0	1	0	1	0	1	0	1	...	(01)

The computation of V and V' terminates on periodic words because there is a finite number of configurations (bounded by the product of the period lenghts of w and w').

The algorithm for periodic clocks is the following: let $w = a.(b)$ and $w' = a'.(b')$ with $|a|_1 = |a'|_1$ and $|b|_1 = |b'|_1$ (it is possible following remark 1). Then $v = c.(d)$ and $v = c'.(d')$ with

1. $|c| = |c'| = \max([a]_1 - [a']_1) + \max\big((|a| - |a|_1) - (|a'| - |a'|_1)\big) + \Sigma_{i=2}^{|a|_1} \max\big(([a]_i - [a]_{i-1}) - ([a']_i - [a']_{i-1})\big)$,

2. $|d| = |d'| = \max([b]_1 - [b']_1) + \max\big((|b| - |b|_1) - (|b'| - |b'|_1)\big) + \Sigma_{i=2}^{|b|_1} \max\big(([b]_i - [b]_{i-1}) - ([b']_i - [b']_{i-1})\big)$,

3. let $n_j = \max([a]_1 - [a']_1) + \Sigma_{i=2}^{j_1} \max\big(([a]_i - [a]_{i-1}) - ([a']_i - [a']_{i-1})\big)$, if $[a]_j \leq [a']_j$ (the other case is symmetrical), then:

[6] In fact, there is always an infinite set of maximal words for $\leq_{SS}$ of the form a `on` $w = a'$ `on` $w' \leq_{SS}$.

$c[n_j + k] = 1$ for $k = 0...[a]_j - [a]_{j-1}$
$c'[n_j + k] = 1$ for $k = 0...[a']_j - 1$ and $k = [a]_j - [a]_{j-1}$
$c'[n_j + k] = 0$ for $k = [a']_j - 1...[a]_j - [a]_{j-1}] - 1$

Theorem 1 (greedy-unification) *Let w and w' be two infinite periodic binary words, and let (v, v') be the results of the previous algorithm on (w, w'). Word $m = v$ on $w = v'$ on w' is minimal for the precedence relation $\preceq$ over all sub-samplings of w and w'.*

Proof. *Notice $\preceq$ is associated with a complete lattice on infinite binary words, but this lattice is* not *complete on periodic words.*[7] *This choice is also maximal for $\leq_{SS}$ since spurious 0s would contradict the minimality for $\preceq$.*

The unification theorem is proven inductively on the position of the p-th 1 in m. Consider an infinite periodic binary word a of the form $a = u$ on $w = u'$ on w'. By construction of m, $[m]_1 = \min([w]_1, [w']_1)$, hence $[m]_1 \leq [a]_1$. Assume $[m]_p \leq [a]_p$ for some $p \geq 1$. Observe that between two consecutive 1s in m, the associated subword of either v or v' is a sequence of 1s; therefore, either $[m]_{p+1} - [m]_p = [w]_{p+1} - [w]_p$ or $[m]_{p+1} - [m]_p = [w']_{p+1} - [w']_p$. Since have $w \preceq a$ and $w' \preceq a'$ ($\leq_{SS}$ is a reversed sub-order of $\preceq$), this proves $[m]_{p+1} \leq [a]_{p+1}$, hence $m \preceq a$ by induction on p.

As a corollary, this theorem shows that sub-samples of any finite set of infinite periodic binary words have a complete lower semi-lattice structure for $\preceq$. □

Back to the downscaler, the unification yields

$$(1) \texttt{ on } (10100100) \texttt{ on } 0^{3600}.(1) \texttt{ on } (101001001) = (100001000000010000000100).$$

On this example, the unification is complete, since all equations are of the form $a = a'$ on w'.

Yet the result is *not* equal to the clock constraint (100000). The downscaler is thus rejected in a pure periodic synchronous calculus. This is the reason why we introduce the *relaxed* notion of *synchronizability*, the main contribution of this paper. This will be develop in section 4.

To end this section, we define the synchronous semantics of the language.

3.4.3 Denotational Semantics over Clocked Streams

We first give our core language 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 $\sqsubseteq$ be the prefix order over

[7] Neither upwards nor downwards, even for the restriction to periods with at least one 1.

sequences, i.e., $x \sqsubseteq y$ if x is a prefix of y. The ordered set $D = (T^\infty, \sqsubseteq)$ is a cpo. If D_1 and D_2 are cpo, then $D_1 \times D_2$ is a cpo with the coordinate-wise order. $[D_1 \to D_2]$ as the set of continuous functions from D_1 to D_2 is also a cpo by taking the pointwise order. If f is a continuous mapping from D_1 to D_2, we shall write $\mathit{fix}\ f = lim_{n\to\infty} f^n(\epsilon)$ for the smallest fix point of f (Kleene theorem).

We define the set *Clocked-Stream*(T) of *clocked sequences* as the set of finite and infinite sequences of elements over the set $T_\perp = T + \{\perp\}$.

$$\begin{array}{lcl} \texttt{bool} & = & \{0,1\} \\ T_\perp & = & T + \{\perp\} \\ \mathit{Clocked\text{-}Stream}(T) & = & (T_\perp)^\infty \end{array}$$

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 can define the function *clock* from clocked sequences to boolean sequences:

$$\begin{array}{lcl} \mathit{clock}(\epsilon) & = & \epsilon \\ \mathit{clock}(\perp.s) & = & \mathit{false}.\mathit{clock}(s) \\ \mathit{clock}(x.s) & = & \mathit{true}.\mathit{clock}(s) \end{array}$$

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 7. We use the mark # to distinguish the syntactic construct (e.g., `fby` from its interpretation as a stream transformer).

- 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, $\texttt{const}^\#\ i\ s$ denotes a constant stream with stream clock s ($\mathit{clock}(\texttt{const}^\#\ i\ s) = s$). The base clock is an infinite sequence made of the true values.

- In case of binary operator, we impose that the two arguments be synchronous (together present or together absent) and the purpose of the clock calculus is to ensure statically that no other case may arrive (otherwise, some buffering is necessary).

- `fby` is the unitary delay: it conses the head of its first argument to its second argument. The arguments and the result of `fby` must be on the same clock. `fby` corresponds to a two-state machine: while the two arguments are absent, it emits nothing and stays in its initial state ($\texttt{fby}^\#$). When both are present, it emits its first argument and goes into the new state ($\texttt{fby1}^\#$) 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).

$$
\begin{array}{lcl}
\texttt{const}^{\#}\ i\ true.s & = & i.\texttt{const}^{\#}\ i\ s \\
\texttt{const}^{\#}\ i\ false.s & = & \bot.\texttt{const}^{\#}\ i\ s \\
\texttt{base}^{\#} & = & true.\texttt{base}^{\#} \\
 & & \\
op^{\#}(s_1, s_2) & = & \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(\bot.s_1, \bot.s_2) & = & \bot.op^{\#}(s_1, s_2) \\
op^{\#}(v_1.s_1, v_2.s_2) & = & (v_1\ op\ v_2).op^{\#}(s_1, s_2) \\
 & & \\
\texttt{fby}^{\#}(\epsilon, s) & = & \epsilon \\
\texttt{fby}^{\#}(\bot.s_1, \bot.s_2) & = & \bot.\texttt{fby}^{\#}(s_1, s_2) \\
\texttt{fby}^{\#}(v_1.s_1, v_2.s_2) & = & v_1.\texttt{fby1}^{\#}(v_2, s_1, s_2) \\
\texttt{fby1}^{\#}(v, \epsilon, s) & = & \epsilon \\
\texttt{fby1}^{\#}(v, \bot.s_1, \bot.s_2) & = & \bot.\texttt{fby1}^{\#}(v, s_1, s_2) \\
\texttt{fby1}^{\#}(v, v_1.s_1, v_2.s_2) & = & v.\texttt{fby1}^{\#}(v_2, s_1, s_2) \\
 & & \\
\texttt{when}^{\#}(\epsilon, c) & = & \epsilon \\
\texttt{when}^{\#}(\bot.s, c) & = & \bot.\texttt{when}^{\#}(s, c) \\
\texttt{when}^{\#}(v.s_1, 1.s_2) & = & x.\texttt{when}^{\#}(s_1, s_2) \\
\texttt{when}^{\#}(v.s_1, 0.s_2) & = & \bot.\texttt{when}^{\#}(s_1, s_2) \\
 & & \\
\texttt{merge}^{\#}(s_1, s_2, s_3) & = & \epsilon \text{ if } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(1.s_1, v.s_2, \bot.s_3) & = & x.\texttt{merge}^{\#}(s_1, s_2, s_3) \\
\texttt{merge}^{\#}(0.s_1, \bot.s_2, v.s_3) & = & y.\texttt{merge}^{\#}(s_1, s_2, s_3) \\
 & & \\
\texttt{not}^{\#}1.s & = & 0.\texttt{not}^{\#}s \\
\texttt{not}^{\#}0.s & = & 1.\texttt{not}^{\#}s \\
 & & \\
\texttt{on}^{\#}(1.s, 1.c) & = & 1.\texttt{on}^{\#}(s, c) \\
\texttt{on}^{\#}(1.s, 0.c) & = & 0.\texttt{on}^{\#}(s, c) \\
\texttt{on}^{\#}(0.s, c) & = & 0.\texttt{on}^{\#}(s, c)
\end{array}
$$

Figure 7: Semantics for the core primitives

- The definition of `merge` states that one branch must be present when the other is absent.
- We end with the interpretation of some boolean operations over periods. The other operations (e.g., `or` and `&`) follow the same principle.

It is easy to check that the above primitives are continuous functions from clocked sequences to clocked sequences.

The semantics is given to expressions which have passed the clock calculus. We define the interpretation of clock types as the following:

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

In order to take away causality problems (which are treated by some dedicated causality 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 8.

4 N-Synchronous Language

The downscaler example highlights a fundamental problem to 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 equal".

4.1 Synchronizability

We introduce an equivalence relation to characterize this resynchronization concept for any arbitrary clocks (not necessarily periodic).

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

Figure 8: Synchronous Data-flow Semantics over Clocked Sequences

Definition 2 *Let us consider two clocks clk_1 and clk_2. We say that the clk_i are* synchronizable *and we denote $clk_1 \bowtie clk_2$ iff there exists $d, d' \in \mathbb{N}$ such that $clk_1 \prec 0^d.clk_2$ and $clk_2 \prec 0^{d'}.clk_1$. It means that we can delay clk_1 by d' ticks so that the 1 of clk_2 occur before the 1 of clk_1 and conversely.*

Example 1

1. *$1(10)$ and (01) are synchronizable;*
2. *$11(0)$ and (0) are not synchronizable;*
3. *(010) and (10) are not synchronizable since there are too many reads or too many writes (infinite buffer).*

It means that the n-th 1 of clk_1 is at a bounded distance from the n-th 1 of clk_2. Another consequence is the following property.

Properties 2 *Let us consider two clocks clk_1 and clk_2. If $clk_1 \bowtie clk_2$ then there exists two synchronous processes, called buffers b_1 and b_2 such that $b_1(clk_1) = 0^d.clk_2$ and $b_2(clk_2) = 0^{d'}clk_1$. The notation $b_1(clk_1)$ means that we synchronize clk_1 with b_1 and the synchronized product is the clock $0^d.clk_2$. The storage of b_1 and b_2 is bounded by d and d' respectively.*

Proof. *A buffer of size $d + d'$ is enough for both cases. Indeed, each 1 written by clk_1 will be read at most d' ticks later. In the worst case, d' consecutive 1 are stored in the buffer and then read.* □

Bounds d and d' are not of practical interest; we will show how to compute optimal buffer sizes in the section 4.2.

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

Proposition 4 *Consider two periodic clocks $ck = u(v)$ and $ck' = u'(v')$. Clocks ck and ck' are* synchronizable *— denoted by $ck \bowtie ck'$ — if and only if*

$$\frac{|v|_1}{|v'|_1} = \frac{|v|}{|v'|}.$$

In other words, $ck \bowtie ck'$ means ck and ck' have the same number of 1s in (v) and (v'), hence the same asymptotic production rate. It also means the n-th 1 of ck is at a bounded distance from the n-th 1 of ck'.

Proof. *From remark 1, we consider $clk_1 = u(v)$ and $clk_2 = u'(v')$ with $|u| = |u'|$ and $|v| = |v'|$. Thus $clk_1 = u(v) \bowtie clk_2 = u'(v')$ iff there exists d, d' such that $\forall w \sqsubseteq clk_2[1..|u|+|v|+d]$, $w' \sqsubseteq 0^d.clk_2[1..|u|+|v|]$, $|w| = |w'| \Rightarrow |w|_1 \geq |w'|_1$ and $\forall w \sqsubseteq 0^{d'} clk_1[1..|u|+|v|]$, $w' \sqsubseteq clk_2[1..|u| + |v| + d'], |w| = |w'| \Rightarrow |w|_1 \geq |w'|_1$. It is sufficient to cover the prefixes of finite length $\leq |u| + |v| + max(d + d')$.*

The 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 clk_1 and clk_2. Let us denote $a = |v|_1 - |v'|_1$, then v^n has na 1 more than v'^n. Thus $v^n \prec 0^{f(n)}.v'^n$ where $|v^n| \geq f(n) \geq na$ and $f(n)$ is minimal in the sense that $v^n \not\prec 0^{f(n)-1}.v'^n$. It entails that $(v) \prec 0^{lim f(n)}.(v')$ and thus there are not synchronizable.

Conversely, let us assume that $|v|_1/|v'|_1 = |v|/|v'|$. Since u and u' are finite, there are almost synchronizable. We have $1^r.u \prec 0^p.u'$ and $1^k.u' \prec 0^q.u$ with $r = max(0, |u'|_1 - |u|_1)$, $k = max(0, |u|_1 - |u'|_1)$. (v), $p = min\{l \mid l \leq |u| + r \wedge 1^r.u \prec 0^l.u'\}$ and $q = min\{l \mid l \leq |u'| + \wedge 1^k.u' \prec 0^l.u\}$. (v') are also synchronizable, thus $(v) \prec 0^m.(v')$ and $(v') \prec 0^n.(v)$. Then $clk_1 \prec 0^{p+m+r|v|}.clk_2$ and $clk_2 \prec 0^{q+n+k|v'|}.clk_1$. We delay $r|v|$ more times because there is at least one 1 in each pattern (v). □

4.2 Relaxed Clock Calculus

Predicate $H \vdash e : ct$ states that expression e has clock ct in clock environment H.

The clock calculus is modified in two ways:

1. a sub-typing [23] rule (SUB) is added to the clock calculus to permit the automatic insertion of a finite buffer in order to synchronize clocks;

2. rule (CONSTRAINT) is extended into a sub-typing rule to allow automatic insertion of a bounded delay.

Notations We first recall some basics on sub-typing. The presence of sub-typing does not change how the programs are evaluated. Sub-typing is just a manner of obtaining more flexibility in typing terms.

Definition 3 (Sub-Types [23]) *We say that S is a* sub-type *of T, denoted by $S <: T$, if every term of type S can be used in context where terms of type T are expected.*

We add then in the typing rules, the sub-typing rule, called *subsumption* rule, stating that in a context Γ, an expression e of type S can be used with the type T:

$$\text{(T-SUB)}\quad \frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T}$$

The relation $<:$ must be reflexive and transitive, and may also have a Top.

Let us define relation $<:$ defined by:

$$pe_1 <: pe_2 \Longleftrightarrow pe_1 \bowtie pe_2 \wedge pe_1 \preceq pe_2.$$

This is a partial order and a lattice, since it is the quotient of the lattice structure of $\preceq$ over $\bowtie$.

The Sub-Typing Rule Relation $<:$ defines a sub-typing rule (SUB) on clock stream types:

$$\text{(SUB)}\quad \frac{P, H \vdash_s e : ck \texttt{ on } pe_1 \quad pe_1 <: pe_2}{P, H \vdash_s e : ck \texttt{ on } pe_2}$$

This sub-typing rule is a standard assumption rule, and all classical results on sub-typing apply [23] because of the lattice structure of the $<:$ relation.

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*.

For example, the following program is rejected in the previous clock calculus since clock α `on` (01) cannot be unified with clock α `on` $1(10)$, assuming x has some clock α.

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

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

1. (NODE): the signature of f is of form $f : \alpha \rightarrow ck_f$;
2. (WHERE) and (+): calling ck_{e_1} and ck_{e_2} the respective (unknown) clocks of e_1 and e_2, we get the constraints $ck_{e_1} <: ck_f$ and $ck_{e_2} <: ck_f$;
3. (WHEN): we get $ck_{e_1} = \alpha$ `on` (01) and $ck_{e_2} = \alpha$ `on` $1(10)$;
4. (SUB): ck_{e_1} and ck_{e_2} are synchronizable since $(01) \bowtie 1(10)$; we may thus choose $ck_f = \alpha$ `on` (01) since $1(10) \preceq (01)$.

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

Considering the downscaler example, this sub-typing 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 (CONSTRAINT) is relaxed into the following sub-typing rule:

$$\text{(CONSTRAINT)} \ \frac{P, H \vdash_s e_1 : ck \ \texttt{on} \ pe_1 \quad P, H \vdash_s e_2 : ck \ \texttt{on} \ pe_2 \quad pe_1 <: 0^d pe_2}{P, H \vdash_s e_1 \ \texttt{at} \ e_2 : ck \ \texttt{on} \ 0^d pe_2}$$

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

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

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

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

Finally, $f : \forall a.a \ \rightarrow \ a \ \texttt{on} \ 0(1001)$. 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 rule (SUB). The rules are given in figure 9.

Thus, starting from a standard clock calculus whose purpose is to reject non-synchronous program, we extend it with a *sub-typing* rule 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.

4.3 Resolution and Buffer Size Inference

Each time the (CONSTRAINT) rule is applied, the resolution algorithm consists in computing a possible value of d. This algorithm is syntax directed, and always chooses to minimize delay insertion. If no delay is needed, $d = 0$ is the chosen solution. In general, the algorithm chooses the minimal value for d: $\text{delay}(ck, ck') = \min\{l \mid ck \preceq 0^l ck'\}$. Note that in general, $\text{delay}(ck, ck') \neq \text{delay}(ck', ck)$.

$$\text{(IM)}\quad P, H \vdash_s i : ck \qquad\qquad \text{(INST)}\ \frac{ct \leq H(x)}{P, H \vdash_s x : ct}$$

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

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

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

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

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

$$\text{(WHERE)}\ \frac{P, H, x : ct \vdash_s e : ct \quad P,\ H, x : ct \vdash_s e' : ct'}{P, H \vdash_s e' \texttt{ where } x = e : ct'}$$

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

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

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

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

$$\text{(SUB)}\ \frac{P, H \vdash_s e : ck \texttt{ on } pe_1 \quad pe_1 <: pe_2}{P, H \vdash_s e : ck \texttt{ on } pe_2}$$

$$\text{(CONSTRAINT)}\ \frac{P, H \vdash_s e_1 : ck \texttt{ on } pe_1 \quad P, H \vdash_s e_2 : ck \texttt{ on } pe_2 \quad pe_1 <: 0^d pe_2}{P, H \vdash_s e_1 \texttt{ at } e_2 : ck \texttt{ on } 0^d pe_2}$$

Figure 9: The Relaxed Clock Calculus

Proposition 5 *Consider two synchronizable periodic clocks w and w'. The delay to synchronize w with an imposed clock w' can be automatically computed by the formula*

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

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: 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, $w \not\preceq v'$.*

□

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 $0^{9600}.(100001000000010000000100) \preceq 0^d(100000)$. 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.

Each time the (SUB) rule is applied, the resolution algorithm computes the minimal synchronizable clock (according to partial order $\preceq$). Given periodic clocks ck and ck', the minimal synchronizable clock is $\alpha = ck \sqcup ck'$.

Proposition 6 *Consider two synchronizable periodic clocks ck and ck'. The minimal buffer to allow communication from clock ck to clock $ck \sqcup ck'$ is of size*

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

Communication from ck to $ck \sqcup ck'$ is called n-synchronous.

This size is effectively computable thanks to Remark 1.

This is a lower bound on the minimal size, since n is the maximal number of pending writes which appear before their matching reads. It is also an actual minimal size, since it is possible to implement a size n buffer with n registers, as described in Section 5.

For the simplified downscaler, buffer size n is equal to 1, since clock $0^{9600}.(1000010000000 10000000100)$ may at most take one advanced tick with respect to clock $0^{9603}(100000)$. For the real downscaler, we automatically computed $n = 880$.

Example 2 *Let us consider again the following program:* `node` $f(x) = y$ `where` $y = (x$ `when` $(01)) + (x$ `when` $1(10))$. *The addition is realized on clock* $(01) \sqcup 1(10) = (01)$. *Practically, we need to introduce one buffer of size 1 between x* `when` $1(10)$ *and $+$ to perform the operation. The relaxed addition is depicted in the Figure 10. The channel x* `when` (01) *remains unchanged, while there is a buffer (in green) in the x* `when` $1(10)$ *channel which transforms the clock (in blue) into the supremum* (01).

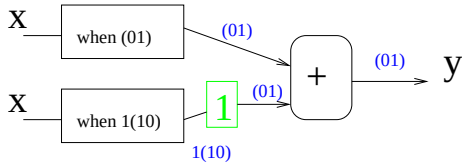


Figure 10: Example of Relaxed Addition

Example 3 *Let us consider a complete system*

$$\texttt{node}\ f(x) = y\ \texttt{where}\ y = (x\ \texttt{when}\ (10)) + (x\ \texttt{when}\ 1(10))$$
$$\texttt{node}\ g(i, z) = x\ \texttt{where}\ x = i + z$$
$$\texttt{node}\ \mathit{stutter}(y) = z\ \texttt{where}\ z = \texttt{merge}\ (10)\ y\ 0\ \texttt{fby}\ z\ \texttt{whenot}\ (10)$$

This system is depicted in the figure 11.

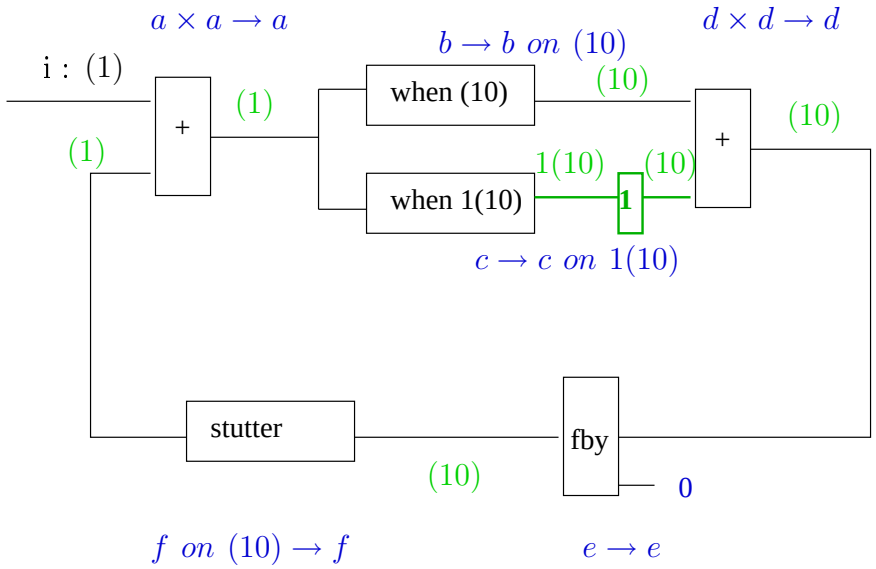


Figure 11: Accepted Program

The process signatures (in blue in the Figure) are computed following the rules. The addition is an operator and by the rule (OP) has the signature $\alpha \times \alpha \rightarrow \alpha$. The fby operator is conservative in clock type. Now we need to compute the signature of stutter: $ct \rightarrow ct'$ where $ct' = \alpha$ and $ct \subseteq \alpha\ on\ (10)$. Thus the system is accepted is the following system has a solution. The solution is computed using the unification (in green in the Figure 11).

$$\begin{cases} a <: b \\ a <: c \\ b \texttt{ on } (10) <: d \\ c \texttt{ on } 1(10) <: d \\ d <: e \\ e <: f \texttt{ on } (10) \\ f <: a \\ (1) <: a \end{cases}$$

$a = b = c = f = (1)$ *and* $e = d = a$ `on` (10) *is a solution. It means that the program is accepted. On the opposite, the system where* f *is replaced by:*

$$\texttt{node } f(x) = y \texttt{ where } y = (x \texttt{ when } (01)) + (x \texttt{ when } 1(10))$$

This system is represented in the figure 12.

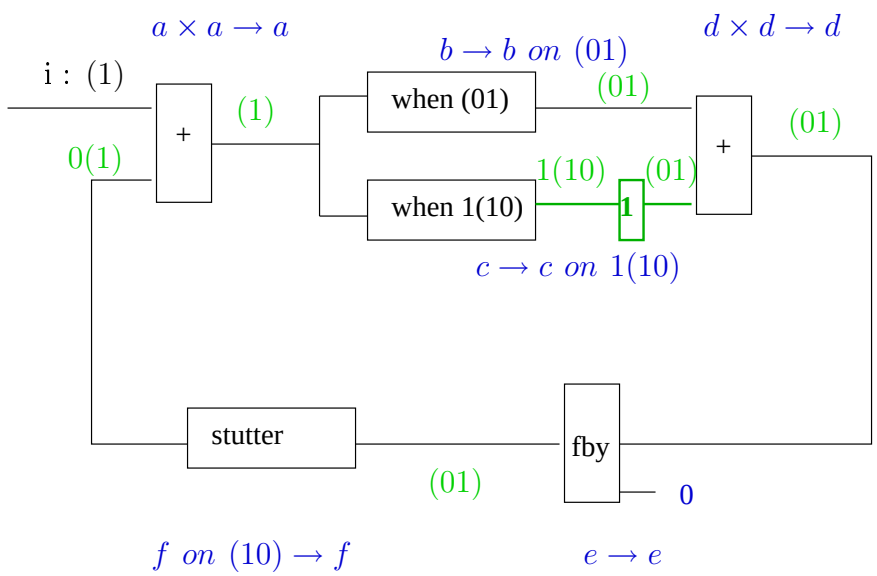


Figure 12: Rejected Program

$$\begin{cases} a <: b \\ a <: c \\ b \texttt{ on } (01) <: d \\ c \texttt{ on } 1(10) <: d \\ d <: e \\ e <: f \texttt{ on } (10) \\ f <: a \\ (1) <: a \end{cases}$$

$a = b = c$, $e = d = a \texttt{ on } (01) = (a \texttt{ on } 0(1)) \texttt{ on } (10)$ *and thus* $f = a \texttt{ on } 0(1) = a$. *Thus the only solution is* $a = \alpha \texttt{ on } (0)$. *And in this case, the clock calculus accepts the program but the only solution is the synchronous program idle (or false). Since there is an imposed constraint of clock input to be equal to* (1), *the program is rejected.*

5 Code Generation and Buffer Insertion

The rules (SUB) and(CONSTRAINT) identify where a buffer should be used to synchronize a producer and a consumer. The size of this buffer can be computed statically, thanks to the $\preceq$ relation. Such a buffer being itself a synchronous program (yet accepted by the original clock calculus), this means that the whole program can be transformed into a 0-synchronous program.

5.1 Buffer Construction

One may need to effectively generate synchronous code for automatically inserted buffers, using as many registers as required by the size inference rule defined in the previous section. There are two main solutions.

- The first one is mostly theoretical and helps to understand the combinatorics intrinsic to n-synchronous communication between periodic clocks. It reduces the n-synchronous program to a (0-)synchronous one with *statically known periodic clocks*, expressible in the original core calculus. Unfortunately, this strategy requires an exponential memory and code size to characterize all control states of multiple communicating FIFO buffers.

- The second one reduces the program to a synchronous one expressible in an classical (non periodic) synchronous calculus. It implements FIFO buffers where the presence or absence of data is captured by dynamically computed clocks. The memory and code size become linear in the total buffer size and appropriate for a practical implementation. Yet static properties about the behavior of the buffers 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 when at most n tokens are sent and not yet received.

5.1.1 Theoretical Construction of a Synchronous Buffer Code

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

`NOP` — $w[j] = 0$ **and** $w'[j] = 0$**:** Nothing happens in the buffer: $clk_i[j] = 0$, $w_i[j] = w_i[j-1]$; 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 $x_1 = \textit{input}$, $\forall i > 2, w_i[j] = w_{i-1}[j-1]$, $w_1[j] = 1$ and $clk_i[j] = 0$.

`POP` — $w[j] = 0$ **and** $w'[j] = 1$**:** Let $p = max(\{0\} \cup \{1 \leq i \leq n | w_i[j-1] = 1\})$. 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 clk_0, setting $clk_i[j] = 0$ for $i > 0$ and $clk_0[j] = 1$, $w_i[j] = w_i[j-1]$. Conversely, if $p > 0$, $\forall i \neq p, clk_i[j] = 0$, $clk_p[j] = 1$, $\forall i \neq p, w_i[j] = w_i[j-1]$ 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** $w'[j] = 1$**:** A `POP` is performed, followed by a `PUSH`, as defined in the two previous cases.

Assuming $w = u.(v)$ and $w' = u'.(v')$ have been written under the lines of remark 1, it is sufficient to conduct this simulation for $|u| + |v|$ cycles to compute periodic clocks w_i and clk_i.

This static simulation of a FIFO can be implemented in a plain synchronous language. Unfortunately, this implementation is impractical because each of the clocks w_i and clk_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 more practical buffer implementations, decoupling the memory management for the FIFO from the combinatorial control space.

5.1.2 Practical Synchronous Buffer Implementation

A n-buffer can be implemented synchronously as a sequence of connected one-buffers in complexity proportional to n.

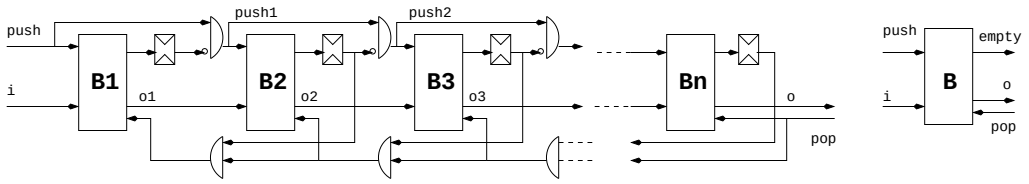


Figure 13: A Synchronous Buffer

A one-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. When 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. Let us give an implementation of this buffer in a strictly synchronous language, LUCID SYNCHRONE [8][8].

```
let buffer1 (push, pop, i) = (empty, o) where
  rec 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
```

The buffer of size n can be constructed by connecting a sequence of one-buffers as shown in figure 13.

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

[8]The distribution and reference manual can be found at `www-spi.lip6.fr/lucid-synchrone`.

5.2 Translation Semantics

Consider the input period pe_1 and the output period pe_2. We apply the (SUB) rule to build a new *buffer* node $\texttt{buffer}_{pe_1,pe_2}(.)$ with clock $ck \texttt{ on } pe_1 \rightarrow ck \texttt{ on } pe_1 \sqcup pe_2$; pe_1 states when a *push* has to be made whereas $pe_1 \sqcup pe_2$ states when a *pop* occurs.

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 (SUB) is applied. 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:

$$\text{(SUB-translation)} \quad \frac{P, H \vdash_s e : ck \texttt{ on } pe_1 \Rightarrow e' \quad pe_1 <: pe_2}{P, H \vdash_s e : ck \texttt{ on } pe_2 \Rightarrow \texttt{buffer}_{pe_1,pe_2}(e')}$$

Other rules are simple morphisms.[9]

5.3 Correctness and Completeness

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 clocks.

Theorem 2 (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 rules underlying $\vdash_s$ judgments: classical subtyping theory [23, 1] 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 these buffers' signatures are tailored to the resynchronization of inconsistent clocks. This ensures the translated program is synchronous.

Completeness is an open issue. We could not yet prove that the unification scheme leads to consistent n-synchronous clocks for any n-synchronous program. Intuitively, if every pair of communicating clocks is synchronizable, the only reason to reject a program would be that impossible constraints arise across cycles of the process network — accumulated delays inconsistent with asymptotic clock rates. Indeed, the unification method, as well as the (SUB)

[9] Notice the (CONSTRAINT) rule shifts a clock constraint imposed by the programmer; this rule may indirectly lead to the insertion of a synchronization buffer, triggering the (SUB) rule.

and (CONSTRAINT) rules are designed to minimize delay insertion, hence to avoid spurious delay accumulation. We are thus confident about the clock calculus's ability to accept most programs (with periodic clocks) that can be translated to a strictly (0-)synchronous framework; yet a formal completness result is not available.

6 Related Works

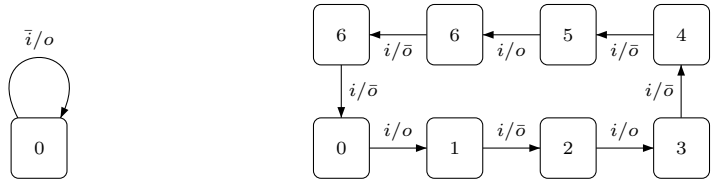
6.1 An alternative approach: i/o automata

The notion of relaxed synchrony can be recast in an equivalent approach based on synchronous Mealy machines [21]. We refer here to an approach used for modeling a signal processing component designed on a chip [2]. In this paper, the authors want to prove that two Soc designs are equivalent and thus use a minimization on automata in order to find an optimized comparison.

We consider the finite alphabet $\Sigma = \{i/o, i/\bar{o}, \bar{i}/o, \bar{i}, /\bar{o}\}$. The meaning of Σ is the following:

- the symbol i stands for an input occurs,
- the symbol o stands for an output occurs, so that for instance the event i/o means that an input and an output occur simultaneously,
- $\bar{i}$ stands for no input occurs and the symbol $\bar{o}$ stands for no output occurs.

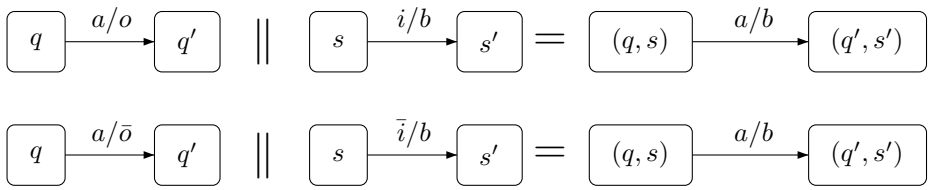
For modeling the system, we depict each component by an i/o-automaton and this can be automatically realized by describing the dependency function given by the processing algorithm. We represent the input and the horizontal filter by their i/o-automaton.



$\mathcal{A}_0 = input$ $\mathcal{A}_1 = horizontal\ filter$

It is easy to see that the language recognized by an automaton $L \subseteq \Sigma^*$ can be seen as $L_1 \times L_2 \subseteq \{0,1\}^* \times \{0,1\}^*$ with $l \leftrightarrow 1$ and $\bar{l} \leftrightarrow 0$ with $l = i, o$. Then we notice that $L_1 \times L_2$ corresponds to the signature $L_1 \rightarrow L_1\ on\ L_2$ defined in the previous section.

The synchronization of i/o-automata is defined as follows: if two components are connected, if the first emits an output thus the second needs to read it, otherwise we enter in a deadlock state. If the first does not emit an output, thus the second must require an input. Let $a, c = i$ or $\bar{i}$ and $b, d = o$ or $\bar{o}$, the synchronization is represented by:



There is again a strong parallel with the definition of periodic clocks. Let us consider two i/o-automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with $L(\mathcal{A}_1) \equiv L_1^1 \times L_2^1$ and $L(\mathcal{A}_2) \equiv L_1^2 \times L_2^2$. Then $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ recognizes $L(\mathcal{A}) \equiv L_1 \times L_2$. We have $L_1 \rightarrow L_1 \; on \; L_2 = L_1^1 \rightarrow (L_1^1 \; on \; L_2^1 \; on \; (L_1^2 \; on \; L_2^2))$. This means that $L_1 = L_1^1$ and $L_2 = L_2^1 \; on \; (L_1^2 \; on \; L_2^2)$.

A buffer of size n is an $n+1$-automaton which can store a data if it is not full and which can deliver a data if it is not empty. At any time, it can push and pop simultaneously.

The computation of the delay and the size boils down to reachability analysis. Our approach with a clock calculus on infinite binary words is directly applicable to a synchronous language and is *a priori* simpler.

6.2 Other approaches

There are a number of approaches for the specification and design of hardware/software systems. Most of them are on 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 [15] or the Cosy project [6]; the use of such tools still requires high expertise from different domains and there is no really universal language that combines in a single framework functional and structural features.

Ptolemy [7] is a rich platform with simulation and analysis tools for the design of embedded streaming systems: it provides the synchronous data-flow (SDF) model of computation. Unlike synchronous languages, SDF graphs are not explicitly clocked: synchrony is a consequence of local balance equations on periodic execution schemes. The SDF model is convenient for the automatic derivation of timing properties [22] 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, 3].

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 [26] is a language for high performance streaming computations that tackles mainly stream-level and algebraic optimization issues.

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.

For this purpose, we identified a class of periodic clocks which proved useful in the context of video processing algorithms. We extend a core data-flow model with this notion of periodic clocks and with a relaxed clock calculus to compose synchronous processes. An implementation in the synchronous language Lucid Synchrone is under way and was applied to a classical video downscaler example.

This work paves the way for numerous extensions. In particular, we would like to define an algebraic framework for n-synchronous clocks based on diadic numbers, in part to benefit from efficient algorithms on rational numbers. We also wish to study the connection between retiming and delay insertion, as a means to express architecture-aware optimizations. A number of applications deal with "jittering" or "bursty" streams that are not strictly periodic; an extension towards more expressive clocks would be beneficial.

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] M. Baclet, R. Pacalet, and A. Petit. Register transfer level simulation. Technical Report LSV-04-10, Lab. Specification and Verification, ENS de Cachan, Cachan, France, 2004.

[3] 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.

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

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

[6] 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 (DAC2000)*, pages pp. 406–409, Los Angeles, CA, USA, June 2000.

[7] 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.

[8] P. Caspi and M. Pouzet. Lucid Synchrone, an ML extension to Lustre. In *Submited for publication.*

[9] 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.

[10] 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.

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

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

[13] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[14] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.

[15] 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.

[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] F. Maraninchi and N. Halbwachs. Compiling ARGOS into boolean equations. In *FTRTFT (Formal Techniques in Real-Time and Fault Tolerant Systems)*, pages 72–89, Uppsala, Sueden, 1996.

[22] 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 25 - 26 2004. ISBN: 90-73461-43-X.

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

[24] Didier Rémy and Jerome Vouillon. Objective ml: A simple object-oriented extension of ml. In *POPL*, pages 40–53, 1997.

[25] Philips Semiconductors. The Nexperia PNX8550 home entertainment engine. `http://www.semiconductors.philips.com/products/nexperia/home/products/hee/#pnx8550`.

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

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

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399