Synchronization of Periodic Clocks [*]

Albert Cohen[1] Marc Duranton[2] Christine Eisenbeis[1]
Claire Pagetti[1] Florence Plateau[3] Marc Pouzet[3]
[1] ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, France
[2] Philips Research Laboratories, Eindhoven, The Netherlands [3] LRI, Paris-Sud University, France

Abstract

We propose a programming model dedicated to real-time video-streaming applications for embedded media devices, including high-definition TVs. This model is built on the synchronous programming model extended with 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.

Such properties are tedious to analyze by hand, due to the combinatorics of video filters, multiple data rates and formats. We show how to extend a core synchronous data-flow language with a notion of periodic clocks, and to 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.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.3.2 [**Language classifications**]: Data-flow languages

General Terms

Performance, Reliability, Languages, Theory

Keywords

High-performance video streaming. Synchronous language. Correctness by construction. Type inference and sub-typing.

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.

General-purpose architectures and compilers are not suitable for the design of real-time *and* high-performance (massively parallel) *programmable* system-on-chip [4]. Achieving a higher compute density and still preserving programmability is a challenge for the choice of an appropriate architecture, programming language and compiler.

Interestingly, the synchronous execution paradigm [1] 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, LUSTRE (SCADE), ESTEREL. It is thus natural to investigate the applicability of such languages for the design of warrantable high-performance systems like high-end video applications.

Due to lack of space, technical details have been omitted. They can be found in the associated research report [5].

2 Motivation

Our main motivating applications are video stream processing for high-definition TV [9]. 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.

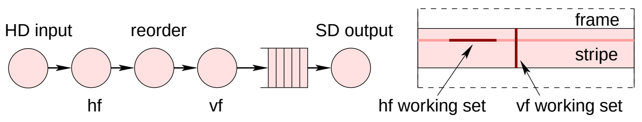

Figure 1. The downscaler

These applications involve a set of scalers resize images in real-time. Our running example is a classical downscaler [4], depicted in Figure 1. It converts a high definition (HD) video signal, 1920×1080 pixels per frame, into a standard definition (SD) output for TV screen, that is 720×480:[1] A horizontal filter — hf — reduces the number of pixels from 1920 down to 720 per line by interpolating packets of 6 pixels. A reordering module — reorder — stores 6 lines of 720 pixels. 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. The embedded system designer is looking for a programming language which *statically* guarantees

[*]This work is partially funded by the French "ACI Sécurité" Alidecs and by an INRIA postdoctoral fellowship.

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

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 before the downstream video processing chain starts receiving pixels;
3. a proof that the system has bounded memory requirements;
4. an evaluation of all memory requirements, including data within all processes and communication buffers.

We will show that, although they do satisfy the four preceding requirements, existing synchronous languages make the implementation of the downscaler 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 [4]): $o_t = \sum_{k=0}^{5} p_{t\times 1920/720+k} \times \mathrm{coef}(k, t \bmod 64)$.

Such filtering functions can easily be programmed in a synchronous data-flow language. Here is the LUCID SYNCHRONE code.

```
let clock c = ok where
  rec cnt = 1 fby (if (cnt = 8) then 1 else cnt + 1)
  and ok = (cnt = 1) or (cnt = 3) or (cnt = 6)
let node hf p = o where
  rec o2 = 0 fby p and o3 = 0 fby o2 and o4 = 0 fby o3
  and o5 = 0 fby o4 and o6 = 0 fby o5
  and o = f(p,o2,o3,o4,o5,o6) when c
val hf :  int => int
val hf :: 'a -> 'a on c
```

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

c	*true*	*false*	*true*	*false*	*false*	*true*	*false*
p	3	4	7	5	6	10	12
o2	0	3	4	7	5	6	10
o	3		14			38	

In the synchronous data-flow 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 boolean stream where $false$ stands for the absence and *true* for the presence. 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 `hf` is $\forall\alpha.\alpha \rightarrow \alpha\ \mathtt{on}\ c$ (printed `'a -> 'a on c`) meaning that for any clock α, if input `p` has clock α, then the output is on a subclock $\alpha\ \mathtt{on}\ c$ defined by the instant where the boolean condition c is true.

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

2.2 The Need for a Relaxed Approach

Real-time constraints on the filters are deduced from the frame rate: the input and output processes enforce that frames are sent and received at 30Hz. This means that HD pixels arrive at $30 \times 1920 \times 1080 = 62,208,000Hz$ — 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. 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 and nothing helps him to compute *automatically* the values **12000** and **880**. Unfortunately, pixels are produced by the downscaler following a periodic but complex event clock. 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 [9].

This paper shows that the principles of synchronous data-flow languages can be relaxed in order to make the computation of process latencies and buffer sizes automatic, using explicit periodic clocks. We introduce the n-synchronous model allowing to compose non stricly synchronous stream as soon as they can be implemented in the ordinary 0-synchronous model with FIFO buffers of size at most n.

3 *N*-Synchronous Streams

We introduce the formal framework to reason about periodic clocks, then apply this framework to the automatic resynchronization of n-synchronous streams.

3.1 Ultimately Periodic Clocks

We consider *infinite binary words*, i.e., words of $(0+1)^\omega$, 0 standing for the boolean value *false* and 1 for the boolean value *true*. We are mostly interested in a subset of the binary words, called *infinite ultimately periodic binary words* or simply *infinite periodic binary words*, defined by the following grammar:

$$w ::= u(v) \quad u ::= \varepsilon \mid 0 \mid 1 \mid 0.u \mid 1.u \quad v ::= 0 \mid 1 \mid 0.v \mid 1.v$$

where $(v) = \lim_n v^n$ is the infinite repetition of *period* v, and u is a prefix of w. We denote by $\mathbb{Q}_2$ the set of infinite periodic binary words; this set coincide with the rational 2-adic numbers [13].

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 following we will consider infinite periodic binary words with an infinite number of 1's, i.e., with period (v) containing at least one 1.

There is an infinite number of representations for an infinite periodic binary word: (0101) is equal to (01) and to $01(01)$. Fortunately, there is a unique, shortest representation of the form $u(v)$: the representation with the shortest prefix and period.

Let $[w]_p$ denote the position of the p-th 1 in w. We have $[1.w]_1 = 1$, $[1.w]_p = [w]_{p-1} + 1$ if $p > 1$, and $[0.w]_p = [w]_p + 1$. Finally, let us define the *precedence* relation $\preceq$ 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, for all $p \geq 1$,

$$[w \sqcup w']_p = \max([w]_p, [w']_p) \quad \text{and} \quad [w \sqcap w']_p = \min([w]_p, [w']_p).$$

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

Finally, if w_1 and w_2 are two infinite periodic binary words, the equality can be decided in quadratic time. Moreover, the pointwise application of boolean operations (e.g., `or`, `not`, `&`) to infinite periodic binary words leads to an infinite periodic binary word. See [5] for further results and details.

Using infinite periodic binary words, we now introduce periodic clocks. A periodic clock is defined by the following grammar:

$$ck ::= \mathtt{base} \mid \alpha \mid ck\ \mathtt{on}\ w \qquad \text{where } w \in \mathbb{Q}_2$$

`base` denotes the base clock and is a shortcut for the infinite periodic binary word (1), α denotes a variable, $ck\ \mathtt{on}\ w$ denotes a *sub-sampled* *clock* of *ck*, where w is itself set on clock *ck*. E.g., (01) `on` $(101) = (010101)$ `on` $(101) = (010001)$.

ck	0	1	0	1	0	1	0	1	0	1	...	(01)
w		1		0		1		1		0	...	(101)
ck `on` w	0	1	0	0	0	1	0	1	0	0	...	(010001)

Formally, `on` is inductively defined as follows: $0.w \text{ on } w' = 0.\,(w \text{ on } w')$, $1.w \text{ on } 0.w' = 0.\,(w \text{ on } w')$ and finally $1.w \text{ on } 1.w' = 1.\,(w \text{ on } w')$

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

PROPOSITION 2 (ON-ASSOCIATIVITY). *Let w_1, w_2 and w_3 be three infinite periodic binary words.*

Then $(w_1 \text{ on } w_2) \text{ on } w_3 \; = \; w_1 \text{ on } (w_2 \text{ on } w_3)$.

3.2 A Synchronous Data-Flow Kernel

We introduce a core data-flow language on infinite streams. Its syntax derives from [6]. Expressions (e) are made of constant streams (i), variables (x), pairs (e,e), local definitions of functions or stream variables (e `where` $x = e$), applications ($e(e)$), initialized delays (e `fby` e) and the following sampling functions: e `when` pe is the sampled stream of e on the periodic clock given by the value of pe, and `merge` is the combination operator of complementary streams (with opposite periodic clocks) in order to form a longer stream; `fst` and `snd` are the classical access functions. e_1 `at` e_2 is used to constraint the clock of e_1 to be on the clock of e_2.

A program is made of a sequence of declarations of stream functions (`let node` $x(x) = e$). Periodic clocks can be combined with boolean operators. Note that pe expression are *static* expressions which can be simplified at compile time into the normal form $u(v)$ of infinite periodic binary words.

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

We can easily program the downscaler in this language kernel. The `main` function consists in composing the various filtering function. The notation `o at (i when (100000))` given by the programmer is a constraint stating that the output pixel `o` must be produced at some clock α `on` (100000), thus 6 times slower that the input clock α.

```
let node hf p = o where
  rec (...)
  and o = f(p,o2,o3,o4,o5,o6) when (10100100)
let node main i = o at (i when (100000)) where rec t = hf i
  and (i1,i2,i3,i4,i5,i6) = reorder t
  and o = vf(i1,i2,i3,i4,i5,i6)
```

The denotational and synchronous (operational) semantics of our core data-flow language as well as its clock calculus are built on classical theory of synchronous languages. The clock calculus can be expressed as a typing problem by asserting judgements of the form $H \vdash e : ct$, meaning that "expression e has *clock type ct* in the clock environment H". This predicate is defined precisely in [5]. Let us illustrate how it works on the downscaler.

1. Suppose that the input `i` has some clock type α_1.
2. The horizontal filter has the following signature, corresponding to the effective synchronous implementation of the process: $\alpha_2 \;\rightarrow\; \alpha_2 \text{ on } (10100100)$.
3. Between the horizontal filter and the vertical filter, the reorder process stores the 5 previous lines in a sliding window of size 5, but has no impact on the clock besides delaying the output until it receives 5 full lines, i.e., $5 \times 720 = 3600$ cycles. We shall give to the buffer operator the clock signature $\alpha_3 \;\rightarrow\; \alpha_3 \text{ on } 0^{3600}(1)$.
4. The vertical filter produces 4 pixels from 9 pixels repeatedly across the 720 pixels of a stripe (6 lines). Its signature (corresponding to the synchronous implementation of the process) is: $\alpha_4 \;\rightarrow\; \alpha_4 \text{ on } (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720})$ To simplify the presentation, we will assume in manual computations that the unit of computation of the vertical filter is a line and not a pixel, hence replace 720 by 1 in the previous signature, yielding: $\alpha_4 \;\rightarrow\; \alpha_4 \text{ on } (101001001)$.
5. Finally, the designer has required (with the `at` construction) that if the input `i` is on clock α_1, then the clock of `o` should be $\alpha_1 \text{ on } (100000)$ — the 6 times sub-sampled input clock — tolerating an additional delay that must automatically be deduced from the clock calculus.

The composition yield the following type constraints: $\alpha_1 = \alpha_2$, $\alpha_3 = \alpha_2 \text{ on } (10100100)$ and $\alpha_4 = \alpha_3 \text{ on } 0^{3600}(1)$.

Resolution of these equations takes the form of a *unification* procedure. Unification in classical clock calculi is syntactical [7]: two clock types ck `on` c and ck' `on` c' can be unified when $c = c'$ and ck and ck' can be unified. In doing this, the sampling construction `on` is not interpreted (property 2 in the general case if nothing is know about c). 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 ck and ck' has to be aware of the properties of the sampling operator `on` between infinite periodic binary words. Details and correctness proofs can be found in [5]. Back to the downscaler, the unification procedure boils down to a simple composition: $((\alpha_1 \text{ on } (10100100)) \text{ on } 0^{3600}(1)) \text{ on } (101001001) = \alpha_1 \text{ on } (10000100000000100000000100)$. 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* and this is the second contribution of this paper.

The downscaler example highlights a fundamental limitation of the synchronous model for programming video streaming applications. 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 synchrony. Our main contribution is a clock calculus to accept the composition of clocks which are "almost equal".

DEFINITION 1. *Consider two infinite periodic binary words $w = u(v)$ and $w' = u'(v')$. w and w' are* synchronizable *— denoted by $w \bowtie w'$ — if and only if $|v|_1/|v'|_1 = |v|/|v'|$.*

In other words, $w \bowtie w'$ means w and w' have the same number of 1s in (v) and (v'), hence the same asymptotic production rate. It also means the n-th 1 of w is at a bounded distance from the n-th 1 of w'. It entails that a process clocked at w can communicate with a process clocked at w' using a *bounded synchronous buffer*, possibly up to a *bounded delay*, and conversely. E.g, $1(10)$ and (01) are synchronizable, as well as (1001) and (0110). However, (010) and (10) are not synchronizable. Technically, the relaxed clock calculus is defined by extending a basic clock calculus defined as a type system relying on clock unification [7] and extended with:

1. a sub-typing rule (SUB) to permit the automatic insertion of a finite buffer in order to synchronize clocks;
2. a constraint rule (CTR) rule to enforce clock constraints up to the automatic insertion of a bounded delay.

Let us define the sub-type relation $<:$ such that:

$$w_1 <: w_2 \iff w_1 \bowtie w_2 \wedge w_1 \preceq w_2.$$

This is a partial order. Relation $<:$ defines a sub-typing rule (SUB) on clock stream types:

$$\text{(SUB)}\quad \frac{H \vdash e : ck \text{ on } w_1 \qquad w_1 <: w_2}{H \vdash e : ck \text{ on } w_2}$$

This is a standard subsumption rule, and all classical results on subtyping apply [12] because of the lattice structure of $<:$.

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*.

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. Clock constraints may be imposed of certain expressions with the following syb-typing rule:

$$\text{(CTR)} \quad \frac{H \vdash e_1 : a \ \mathtt{on}\ w_1 \quad H \vdash e_2 : a \ \mathtt{on}\ w_2 \quad w_1 <: 0^d w_2}{H \vdash e_1 \ \mathtt{at}\ e_2 : a \ \mathtt{on}\ 0^d w_2}$$

Each time the (CTR) rule is applied, the resolution algorithm consists in computing a possible value of the delay 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.

PROPOSITION 3. *Consider two infinite periodic binary words w and w'. The delay to synchronize w with an imposed infinite periodic binary word w' can be automatically computed by the formula*
$delay(w,w') = \max(\max_p([w]_p - [w']_p), 0)$

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
$0^{9600}(1000010000000010000000100) \preceq 0^d(100000)$. For the real downscaler (with fully developed vertical filter signature), we automatically computed that the minimal delay was **12000** to permit communication with the SD output.

4 Code Generation and Buffer Insertion

Each time the (SUB) rule is applied, the resolution algorithm computes the minimal synchronizable clock (according to partial order $\preceq$). Given infinite periodic binary words w and w', the minimal synchronizable infinite periodic binary word is $w \sqcup w'$.

PROPOSITION 4. *Consider two synchronizable infinite periodic binary words w and w'. The minimal buffer to allow communication from w to $w \sqcup w'$ is of size*

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

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

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.

For the simplified downscaler, buffer size n is equal to 1, since clock $0^{9600}(1000010000000010000000100)$ may at most take one advanced tick with respect to clock $0^{9603}(100000)$. For the real downscaler, we automatically computed $n = \mathbf{880}$.

Rule (SUB) tells where a buffer is necessary, 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 Related Works

There are a number of approaches for the specification and design of hardware/software systems. Most of them are graphical tools based on process networks. Kahn process networks (KPN) [10] 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 [8] or the Cosy project [2]; such tools still requires expertise in different domains and there is no universal language that combines functional and structural features in a single framework.

Ptolemy [3] 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 [11] but the lack of clocks weakens its amenability for formal reasoning and correct-by-construction generation of synchronous code, with respect to synchronous languages [1].

6 Conclusion and Perspectives

We proposed an extension of the classical synchronous model 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 is under way and was applied to a classical video downscaler example.

We are working on extending the synchronous data-flow language LUCID SYNCHRONE with these concepts and proving completeness of the type inference. This work paves the way for numerous extensions. In particular, we would like to define an algebraic framework for n-synchronous programs based on 2-adic 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. Finally, a number of applications deal with "jittering" or "bursty" streams that are not strictly periodic; an extension towards more expressive clocks would be beneficial.

7 References

[1] 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.

[2] 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 406–409, Los Angeles, CA, June 2000.

[3] 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.

[4] 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.

[5] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, and M. Pouzet. Synchronizing periodic clocks in kahn networks. Technical Report 5603, INRIA, June 2005. `http://www.inria.fr/rrrt/rr-5603.html`.

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

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

[8] 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.

[9] 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.

[10] 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.

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

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

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