N-Synchronous Kahn Networks

or

Introducing a Relaxed Model of Synchrony for Real-Time Systems

Albert Cohen[1], Marc Duranton[2], Christine Eisenbeis[1], Claire Pagetti[1,4], Florence Plateau[3] and Marc Pouzet[3]

Workshop SYNCHRON, Malta
November 21th, 2005

1: INRIA, Orsay France
2: PHILIPS NatLabs, Eindhoven, The Netherlands
3: University of Paris-Sud, Orsay, France
4: Onera, Toulouse, France

Context

- **video intensive applications** (TV boxes, medical systems).
  tera-operations per second (on pixel components) is typical
- **Ensure three properties:** hard real-time **and** performance **and** safety

Implementations

- Today: specific hardware (ASIC)
- Evolution: multi-clock asynchronous architectures, mixing hardware/software because of costs, variability of supported algorithms

Design and programming tools

- general purpose languages and compilers are not well adapted
- Kahn Process Networks (KPN) is common practice in the field

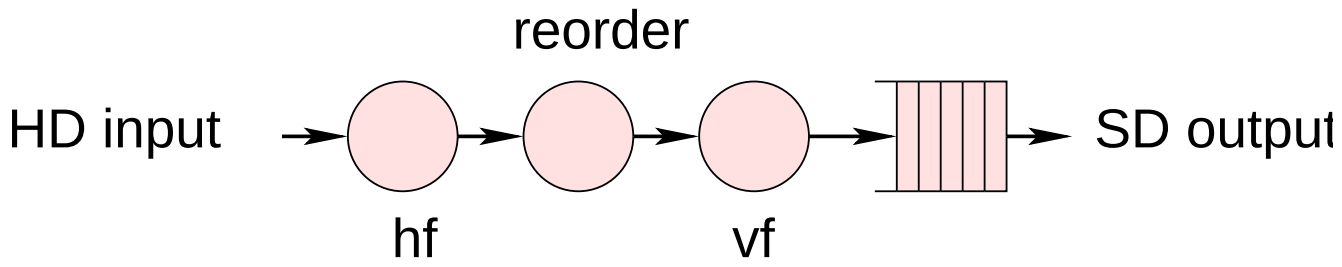
A typical example: the Downscaler

high definition (HD) $\rightarrow$ standard definition (SD)

1920×1080 pixels 720×480

horizontal filter: number of pixels in a line from 1920 pixels downto 720 pixels,

vertical filter: number of lines from 1080 downto 480



Real-Time Constraints

the input and output processes: 30Hz.

HD pixels arrive at $30 \times 1920 \times 1080 = 62,208,000 Hz$

SD pixels at $30 \times 720 \times 480 = 10,368,000 Hz$ (6 times slower)

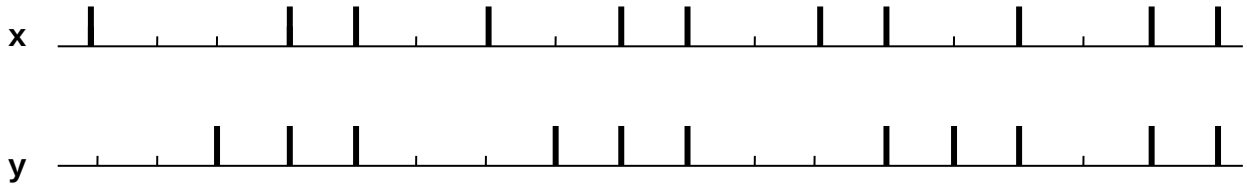
How to program and automatically obtain these rates?

Why not using Synchronous Languages?

- dedicated to hard real-time critical systems
- generation of custom hardware and software system with static guarantees (real-time and resource constraints)
- static analysis, verification and testing tools

But too much restrictive for our video applications

- streams must be synchronous when composed (`x+y` is rejected by the clock calculus)

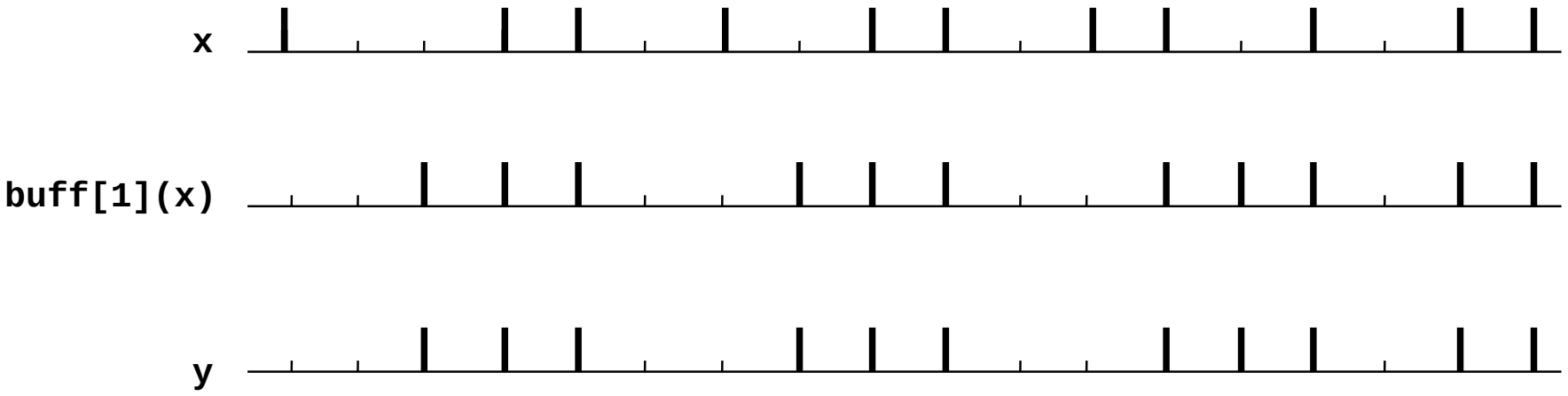


- adding buffer code (by hand) is feasible but hard and error-prone
- can we compute it automatically and obtain regular synchronous code?

we need a relaxed model of synchrony and relaxed clock calculus

N-Synchronous Kahn Networks

- propose a programming model based on a relaxed notion of synchrony
- allows to compose non strictly synchronous programs as soon as they can be made synchronous through the insertion of a bounded buffer



- based on the use of *infinite ultimately periodic clocks* (rational 2-adic numbers)
- a precedence relation between clocks $ck_1 <: ck_2$

Infinite Ultimately Periodic Clocks

Introduce $\mathbb{Q}_2$ as the set of infinite periodic binary words. Coincides with rational 2-adic numbers

$$\begin{aligned} (01) &= 01\ 01\ 01\ 01\ 01\ 01\ 01\ 01\ 01\ldots \\ 0(1101) &= 0\ 1101\ 1101\ 1101\ 1101\ 1101\ 1101\ 1101\ldots \end{aligned}$$

Definition:

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

where $(v) = \lim_n v^n$ is the infinite repetition of *period* v, and u is a prefix of w. $[w]_p$ denote the position of the p-th 1 in w. $|v|_1$ is the number of 1 in v

Remark: There exists an infinite number of representations for an infinite periodic binary word (e.g., (0101) is equal to (01) and to 01(01)) but a unique of the form $u(v)$ with shortest u and v

Causality order and Synchronisability

Precedence relation:

- $w_1 \preceq w_2 \iff \forall p \geq 1, [w_1]_p \leq [w_2]_p$
- $\preceq$ is a partial order which abstracts the causality order between streams
- upper bound $w \sqcup w'$ and lower bound $w \sqcap w'$ as:

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

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

Synchronizability: two infinite binary words w and w' are *synchronizable*, and we write $w \bowtie w'$, iff there exists $d, d' \in \mathbb{N}$ such that $w \preceq 0^d w'$ and $w' \preceq 0^{d'} w$.

Subsumption (sub-typing): $w_1 <: w_2 \iff w_1 \bowtie w_2 \wedge w_1 \preceq w_2$.

Delaying a Clock

Delay the reads after the writes

$$\text{delay}(w, w') = \min\{l \mid w \preceq^l w'\}$$

$w = u(v)$ and $w' = u'(v')$ with $|u| = |u'|$ and $|v| = |v'|$. Then $delay = max(d', 0)$ where

$$d' = max\ \{|w| - |w'| \ \big|\ w.1 \sqsubseteq uv,\ w'.1 \sqsubseteq u'v', |w'|_1 = |w|_1\}$$

For instance if $w_1 = 000001$ and $w_2 = 001000$, then $d = 3$ is reached with $w = 000001$ and $w' = 00$.

Example:

100001	000000	010000	000100
000100	000100	000100	000100

Buffer Size

Consider two synchronizable infinite binary words w and w' such that $w \preceq w'$.

The minimal buffer size n satisfies:

$$n = \max\{(|w|_1 - |w'|_1) \mid w \sqsubseteq uv,\ w' \sqsubseteq u'v';\ |w| = |w'|\}$$

Communication from w to w is called *n-synchronous*

Clock Sampling and Periodic Clocks

$$c \quad ::= \quad w \mid c \texttt{ on } w \qquad\qquad w \in (0+1)^\omega$$

If c is a clock and w is a infinite binary word, then c `on` w denotes a *subsampled clock* of c where w is itself set on clock c.

c `on` w is the clock obtained in advancing in clock w at the pace of clock c. E.g., $00(101)$ `on` $(01) = 0(001)$.

c	0	0	1	1	0	1	1	0	1	1	...	001(101)
half			0	1		0	1		0	1	...	(01)
c `on` *half*	0	0	0	1	0	0	1	0	0	1	...	0(001)

Proposition 1 (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.

Computing with Periodic Clocks

In the case of infinite periodic binary words, precedence relation, synchronizability, equality can be decided in bounded time thank to the following remarks

Synchronizability: Two infinite periodic binary words $u(v)$ and $u'(v')$ are *synchronizable*, noted $u(v) \bowtie u'(v')$ iff they have the same rate, i.e., $\frac{|v|_1}{|v'|_1} = \frac{|v|}{|v'|}$.

Equality: Let $w = u(v)$ and $w' = u'(v')$. We can always write $w = a(b)$ and $w' = a'(b')$ with $|a| = |a'| = max(|u|, |u'|)$ and $|b| = |b'| = \text{lcm}(|v|, |v'|)$

Delays and Buffers: can be computed practically after normalisation

The set of infinite periodic binary words is closed by sampling (`on`), delaying (`pre`) and point-wise application of a boolean operation

$$
\begin{array}{lcl}
w & ::= & u(v) \\
c & ::= & w \mid c \texttt{ on } w \mid \texttt{not } c \mid \texttt{pre } c \mid c \texttt{ or } c \mid \ \ldots
\end{array}
$$

A Synchronous Data-flow Kernel

Reminiscent to `Lustre` and `Lucid Synchrone`

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

The Downscaler

```
let period c = (10100100)

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

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

Clock calculus as a sub-typing system

$$
\begin{array}{lcl}
\sigma & ::= & \forall\alpha.\sigma \mid ct \\
ct & ::= & ct \rightarrow ct \mid ct \times ct \mid ck \\
ck & ::= & \texttt{base} \mid ck\ \texttt{on}\ pe \mid \alpha \\
H & ::= & [x_1 : ct_1, \ldots, x_m : ct_m] \\
P & ::= & [p_1 : pe_1, \ldots, p_n : pe_n]
\end{array}
$$

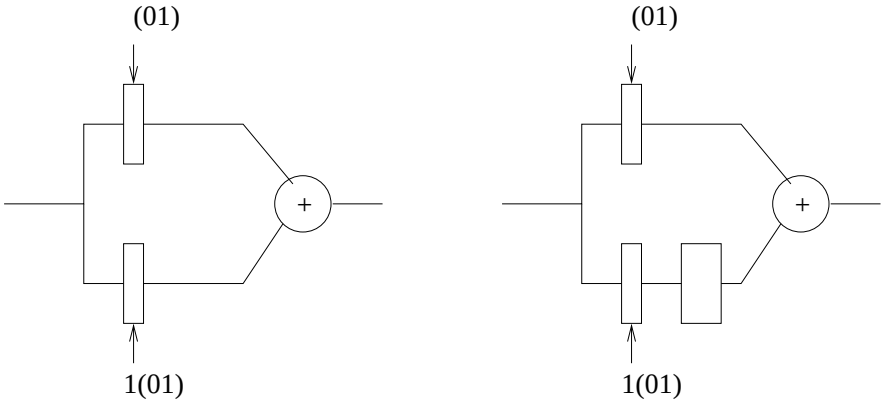
Judgment: $P, H \vdash e : ct$ "expression e receive clock type ct in environments H and P"

Extend the basic clock calculus of a synchronous language with a **sub-typing rule**:

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

defines the synchronization points where buffer code should be inserted

Example



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

(01) and 1(10) can be synchronized using a buffer of size 1. We can apply the rule:

$$\frac{P, H \vdash x \texttt{ when } 1(10) : \alpha \texttt{ on } 1(10) \quad 1(10) <: (01)}{P, H \vdash x \texttt{ when } 1(10) : \alpha \texttt{ on } (01)}$$

and then, classical rules apply and we get the final signature: $f : \forall\alpha.\alpha \rightarrow \alpha \texttt{ on } (01)$

The 1-buffer receive clock type: `buffer[1]`: $\forall\alpha.\alpha \texttt{ on } 1(10) \rightarrow \alpha \texttt{ on } (01)$

The Downscaler

We also may compute the minimal delay to obtain the first pixel

1. input process has some clock α_1
2. `hf`: $\alpha_2 \rightarrow \alpha_2$ `on` (10100100)
3. `reorder`: store 5 lines in a sliding window, thus delaying the output until it receive 5 full lines, i.e., $5 \times 720 = 3600$ cycles.
   We give: $\alpha_3 \rightarrow \alpha_3$ `on` $0^{3600}(1)$
4. `vf`:
$$\alpha_4 \rightarrow \alpha_4 \text{ on } (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720})$$
   Simplification: $\alpha_4 \rightarrow \alpha_4$ `on` (101001001).
5. Finally, suppose that the designer has required the output clock to be of the form α_1 `on` (100000).

The composition of all 5 processes yield the type constraints $\alpha_1 = \alpha_2$, $\alpha_3 = \alpha_2$ `on` (10100100), and $\alpha_4 = \alpha_3$ `on` $0^{3600}(1)$.

Finally, we get for `o`:

$$((\alpha_1 \texttt{ on } (10100100)) \texttt{ on } 0^{3600}(1)) \texttt{ on } (101001001) = \alpha_1 \texttt{ on } 0^{9600}(100001000000010000000100).$$

The result is *not* equal to the clock constraint stating that `o` should be on clock α_1 `on` (100000).

It can be synchronized using a 1-buffer with α_1 `on` $0^{9603}(100000)$.

In the case of the real downscaler, we automatically compute the size **880**.

Algorithm (constraint resolution)

The sub-typing system is not deterministic and is thus not an algorithm

Standard solution:

- apply the (SUB) rule at every program construction
- generate a set of sub-typing constraints $\{ck_1 <: ck'_1, \ldots, ck_n <: ck'_n\}$
- rely on a resolution algorithm

We propose an algorithm which rewrites (simplifies) constraints until we get the empty set constraint.

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

Translation into 0-Synchronous Programs

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

$$\texttt{buffer}_{w,w'} : \forall\alpha.\alpha\ \texttt{on}\ w \rightarrow \alpha\ \texttt{on}\ w'$$

Theorem (correctness): Any well clocked (N-synchronous) program can be transformed into a 0-synchronous program

This is a constructive proof: sub-typing points define where some buffering is necessary (as soon as $w \neq w'$)

Conclusion

- **N-Synchronous Kahn Networks** introduce a relaxed model of synchrony
- extended synchrononous framework: automatic generation of the synchronous buffers which are semantically (as defined by Kahn) guaranteed correct
- **Lustre programs are 0-Synchronous Kahn Networks**
- **Kahn networks are ∞-Synchronous Kahn Networks**
- the SDF model (Lee & Messerschmitt) can be encoded in the model

Current and Future Work

- algebra of the thing, efficient representation, better algorithms
- implementation inside an existing synchronous language
- the completeness theorem is mainly of theoretical interest
- optimization and architecture considerations (buffer size, locality, clock gating)
- express the problem in term of standard optimisation techniques
- forgetting buffering mechanism, these clocks are useful for dealing with several implementation of the same function:
  - parallel version *vs* pipelined version *vs* serial computation
  - going from a version to an other changes clocks
  - how to prove them to be equivalent (or to derive them from the same program)?