Synchrony and Clocks in Kahn Process Networks [a]

Marc Pouzet

Univ. Paris-Sud 11

IUF

ISOR 2008, Algier

November 5, 2008

[a] Joint work with Albert Cohen, Florence Plateau, Louis Mandel

Overview

- Real-time Systems and Synchronous Data-flow Languages
- Synchronous Kahn Process Networks
- Introducing logical time: *clocks*
- Checking synchrony with a dedicated type system: the *clock calculus*
- Relaxed synchrony through buffer communication
- Clock enveloppes and a relaxed clock calculus

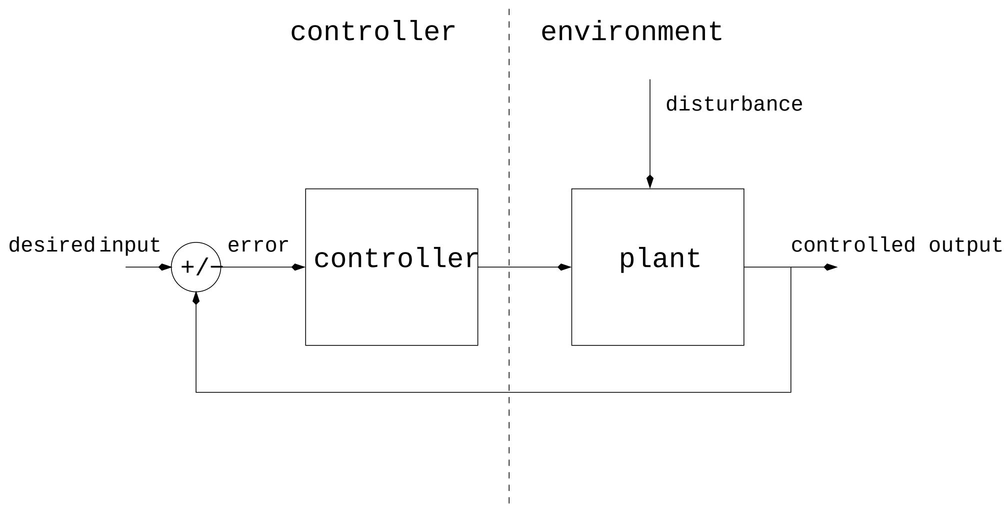
Real-time Systems

Focus on systems which continuously interact with each others.

- with a **physical environment** (e.g., fly-by-wire command, control-engine)
- or **other digital devices** (e.g., phone, TV boxes)

Real time is always **related to the environment** and is not an absolute notion. To ensure safety, think of **“what is the worst case”**?

The environment is often not precisely known: most systems run in ***closed-loop***



How can we program those systems, focusing first on the **functionality**, abstracting some implementation details?

The need for High-level Programming Languages

Conciliate three notions:

- a **formal** (and computable) model of time
  - express deadlines, simultaneous events, etc.
- **parallelism** to describe complex systems from simpler ones
  - control **at the same time** rolling and pitching
  - **closed-loop** systems (the controller and the plant run in parallel)
- **statically guaranty safety properties**
  - **determinism**, dead-lock freedom
  - execution in **bounded time and memory**

Safety is important:

- critical systems: fly-by-wire, braking, airbags, etc.
- properties must be guaranteed statically: **“dynamic” = “too late”**
- build the language on a strong mathematical basis to simplify verification/validation tasks

Synchronous Data-flow Languages

Invented in the 80's to model/program critical embedded software.

The idea of Lustre:

- directly write equations over sequences as **executable specifications**
- provide a **compiler** and static analysis tools to generate code

E.g, the linear filter defined by:

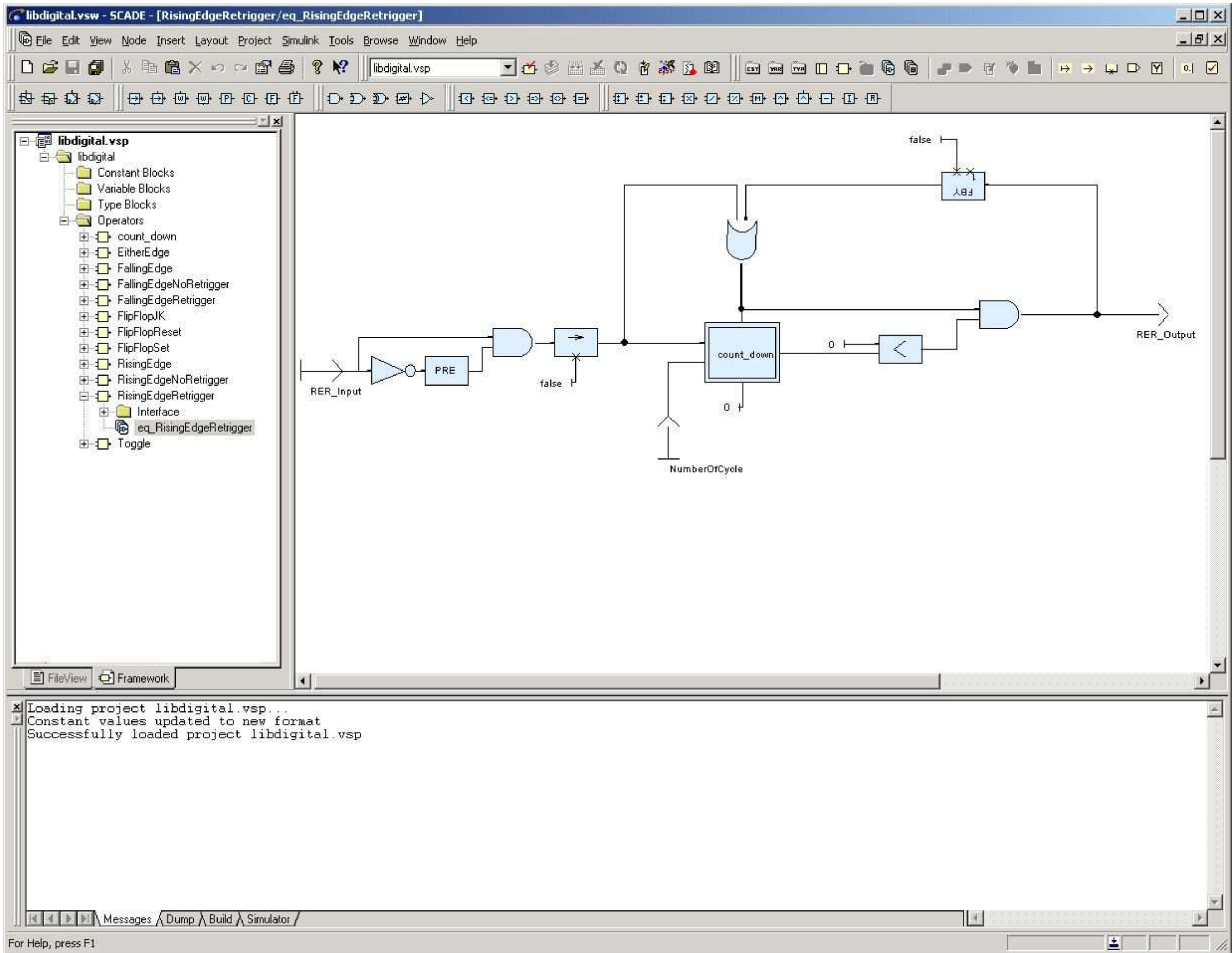
$$Y_0 = bX_0 \ , \ \forall n \ Y_{n+1} = aY_n + bX_{n+1}$$

is programmed by writting the equation:

```
Y = (0 -> a * pre(Y)) + b * X
```

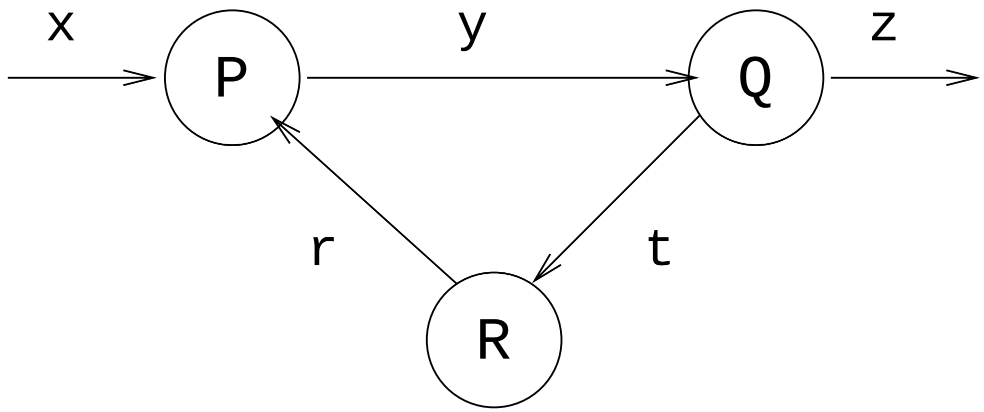
that is, we write **invariants**

An example of a SCADE specification



Kahn Process Networks [IFIP 74]

Kahn answered the following question: What is the semantics of a set of sequential processes communicating through FIFOs (e.g., Unix pipe, sockets)?



- message-based asynchronous communication (`send/wait`) through FIFOs
- reliable channels, bounded communication delays
- waiting on a single channel only. The program:

  `if (A is present) or (B is present) then ...`

  is **forbidden**

Semantics

Domain:

- $V^\infty = V^* + V^\omega$, set of finite and infinite sequences of elements in V.
- V^∞ contains the empty sequence ϵ (bottom element)
- prefix order $\leq_p$: for all $x \in V^\infty, \epsilon \leq_p x$ and for all $v \in V, x, y \in V^\infty$, $x \leq_p y$ iff $v.x \leq_p v.y$
- $(V^\infty, \leq_p, \epsilon)$ is a CPO.

Kahn Principle:

- **a channel** = an history of values $X = x_1, ..., x_n, ... \in V^\infty$
- a process = a function from an *history* of inputs to an *history* of outputs
- **causality**: a process is a **continuous function** ($f(\cup_{i=0}^\infty(x_i)) = \cup_{i=0}^\infty(f(x_i))$)

Interest/Weakness of the model

(+): Simple semantics: a process defines a function (a deterministic system); composition is functional composition; Kleene's fix-point theorem gives meaning to feedback loops

(+): Modularity: a network defines a continuous function; closed by composition and feedback

(+): Time invariance: no explicit time; semantics is invariant through slow-down/speed-up

(+): Distributed asynchronous execution: no need for a centralised scheduler

x	$=$	x_0	x_1		x_2	x_3	x_4	x_5			$\ldots$
$f(x)$	$=$	y_0	y_1		y_2	y_3	y_4	y_5			$\ldots$
$f(x)$	$=$	y_0		y_1	y_2		y_3		y_4	y_5	$\ldots$

A natural model for video streaming applications (TV boxes): Sally (Philips NatLabs), StreamIt (MIT), Xstream (ST-micro) and restricted models *à la SDF* (Ptolemy)

A Small Data-flow Kernel

Consider a small language kernel with basic data-flow primitives

$$
\begin{array}{lcl}
e & ::= & e \ \texttt{fby} \ e \mid op(e, ..., e) \mid x \mid i \\
  &   & \mid \texttt{merge} \ e \ e \ e \mid e \ \texttt{when} \ e \\
  &   & \mid \lambda x.e \mid e(e) \mid \texttt{rec} \ x.e \\
op & ::= & + \mid - \mid \texttt{not} \mid ...
\end{array}
$$

- functions ($\lambda x.e$), application ($e(e)$), fix-point ($\texttt{rec}\ x.e$)
- constant i and variables (x)
- data-flow primitives: $x\ \texttt{fby}\ y$ is the initialized delay; $op(e_1, ..., e_n)$ the point-wise application; sampling operators (`when`/`merge`).

Data-flow Primitives

x	x_0	x_1	x_2	x_3	x_4	x_5
y	y_0	y_1	y_2	y_3	y_4	y_5
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$
x `fby` y	x_0	y_0	y_1	y_2	y_3	y_4
h	1	0	1	0	1	0
x `when` h	x_0		x_2		x_4	
z		z_0		z_1		z_2
`merge` $h\ x\ z$	x_0	z_0	x_2	z_1	x_4	z_3

Sampling:

- if h is a boolean sequence , x `when` h produces a sub-sequence of x
- `merge` $h\ x\ z$ combines two sub-sequences

Kahn Semantics

Define a stream semantics for each data-flow primitive. E.g., if $x \mapsto s_1$ and $y \mapsto s_2$ then the value of $x + y$ is $+^{\#}(s_1, s_2)$

$$
\begin{array}{lcl}
i^{\#} & = & i.i^{\#} \\
\\
+^{\#}(s_1, s_2) & = & \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
+^{\#}(x.s_1, y.s_2) & = & (x + y).+^{\#}(s_1, s_2) \\
\\
\epsilon \, \mathtt{fby}^{\#} \, y & = & \epsilon \\
(x.s_1) \, \mathtt{fby}^{\#} \, s_2 & = & x.s_2
\end{array}
$$

$$
\begin{array}{lcl}
s_1 \, \mathtt{when}^{\#} \, s_2 & = & \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
x.s \, \mathtt{when}^{\#} \, 1.c & = & x.(s \, \mathtt{when}^{\#} \, c) \\
x.s \, \mathtt{when}^{\#} \, 0.c & = & s \, \mathtt{when}^{\#} \, c
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{merge}^{\#} \, c \, s_1 \, s_2 & = & \epsilon \text{ if } s_i = \epsilon \\
\mathtt{merge}^{\#} \, 1.c \, x.s_1 \, s_2 & = & x.\mathtt{merge}^{\#} \, c \, s_1 \, s_2 \\
\mathtt{merge}^{\#} \, 0.c \, s_1 \, y.s_2 & = & y.\mathtt{merge}^{\#} \, c \, s_1 \, s_2
\end{array}
$$

Property: Data-flow operators are continuous functions; a program is a continous functions

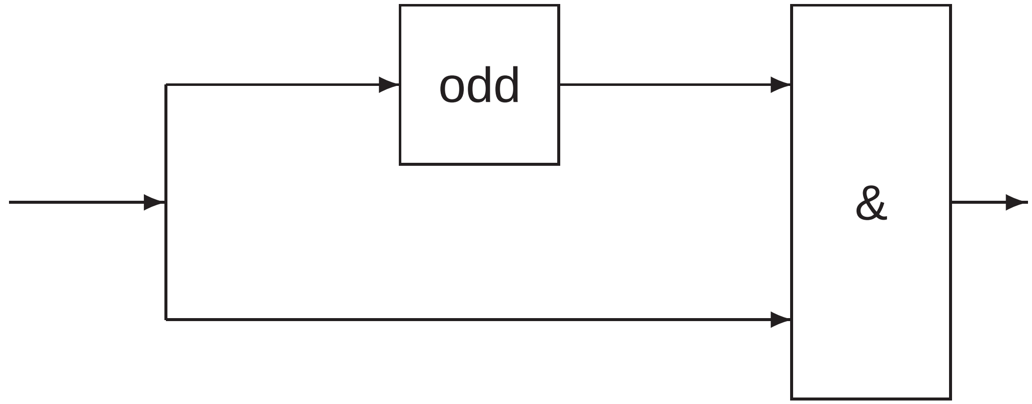
Derived operators:

- $\mathtt{if} \; c \; \mathtt{then} \; x \; \mathtt{else} \; y = \mathtt{merge} \; c \; (x \; \mathtt{when} \; c) \; (x \; \mathtt{when} \; \mathtt{not} \; c)$

Final remark: Up to syntactic details, we can write most Lustre programs.

Synchronisation Issues

What happen when streams are sampled and composed together?



If $x = (x_i)_{i \in \mathbb{N}}$ then $\texttt{odd}(x) = (x_{2i})_{i \in \mathbb{N}}$ and $x \& \texttt{odd}(x) = (x_i \& x_{2i})_{i \in \mathbb{N}}$.

Execution with unbounded FIFOs!

Remarks:

- These programs must be detected and rejected
- each operator is finite-memory through the composition is not: all the complexity (here synchronisation) is hidden in the communication channels
- The Kahn semantics is unable to deal with time, e.g., specify that two event arrive **at the same time**

Synchronous Streams

complete the set of values with an explicit absent value abs. A signal s is a stream.

$$s : (V^{abs})^{\infty}$$

Clock: the clock of a stream x is a boolean stream indicating the instant where x is present

$$\mathbb{B} = \{\mathtt{0}, \mathtt{1}\}$$

$$\mathcal{CLOCK} = \mathbb{B}^{\infty}$$

$$\begin{array}{lcl} \mathtt{clock}\ \epsilon & = & \epsilon \\ \mathtt{clock}\ (abs.x) & = & \mathtt{0}.\mathtt{clock}\ x \\ \mathtt{clock}\ (v.x) & = & \mathtt{1}.\mathtt{clock}\ x \end{array}$$

Clocked Streams:

$$ClStream(V, cl) = \{s | s \in (V^{abs})^{\infty} \wedge \mathtt{clock}\ s \leq_p cl\}$$

Data-flow Primitives

Constant generator:

$$\begin{array}{rcl} i^{\#}(\epsilon) & = & \epsilon \\ i^{\#}(\mathtt{1}.cl) & = & i.i^{\#}(cl) \\ i^{\#}(\mathtt{0}.cl) & = & abs.i^{\#}(cl) \end{array}$$

Pointwise application:

Arguments must be synchronous, i.e., they should have the same clock

$$\begin{array}{rcl} +^{\#}(s_1, s_2) & = & \epsilon \text{ if } s_i = \epsilon \\ +^{\#}(abs.s_1, abs.s_2) & = & abs.+^{\#}(s_1, s_2) \\ +^{\#}(v_1.s_1, v_2.s_2) & = & (v_1 + v_2).+^{\#}(s_1, s_2) \end{array}$$

Partial Definitions

As such, these functions are not total. What does it mean when one element is present and the other is absent?

Restrict the domain:

$$(+) : \forall cl : \mathcal{CLOCK}.ClStream(\texttt{int}, cl) \times ClStream(\texttt{int}, cl) \rightarrow ClStream(\texttt{int}, cl)$$

that is $(+)$ is a function which expect two integer inputs with the same clock cl and return an output with the same clock cl.

These extra conditions are **types**: programs which do not conform to these constraints are rejected.

Remark: Regular types and clock types can be specified separately:

- $(+) : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$ $\leftarrow$ **its type signature**
- $(+) :: \forall cl.cl \times cl \rightarrow cl$ $\leftarrow$ **its clock signature**

In the sequel, we only write the clock signature.

Delays

$$
\begin{array}{lcl}
\epsilon\, \mathtt{fby}^{\#}\, s & = & \epsilon \\
(abs.s_1)\, \mathtt{fby}^{\#}\, (abs.s_2) & = & abs.(s_1\, \mathtt{fby}^{\#}\, s_2) \\
(v.s_1)\, \mathtt{fby}^{\#}\, (w.s_2) & = & v.(\mathtt{fby_1}^{\#}\, w\, s_1\, s_2) \\
\mathtt{fby_1}^{\#}\, v\, \epsilon\, s & = & \epsilon \\
\mathtt{fby_1}^{\#}\, v\, (abs.s_1)\, (abs.s_2) & = & abs.(\mathtt{fby_1}^{\#}\, v\, s_1\, s_2) \\
\mathtt{fby_1}^{\#}\, v\, (w.s_1)\, (v'.s_2) & = & v.(\mathtt{fby_1}^{\#}\, v'\, s_1\, s_2)
\end{array}
$$

As a consequence:

$$\mathtt{fby} : \forall cl.cl \times cl \to cl$$

Sampling

$$
\begin{aligned}
s_1 \,\texttt{when}^{\#}\, s_2 &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
(abs.s)\,\texttt{when}^{\#}\,(abs.c) &= abs.s\,\texttt{when}^{\#}\,c \\
(v.s)\,\texttt{when}^{\#}\,(\texttt{1}.c) &= v.s\,\texttt{when}^{\#}\,c \\
(v.s)\,\texttt{when}^{\#}\,(\texttt{0}.c) &= abs.x\,\texttt{when}^{\#}\,c
\end{aligned}
$$

$$
\begin{aligned}
\texttt{merge}\,c\,s_1\,s_2 &= \epsilon \text{ if one of the } s_i = \epsilon \\
\texttt{merge}\,(abs.c)\,(abs.s_1)\,(abs.s_2) &= abs.\texttt{merge}\,c\,s_1\,s_2 \\
\texttt{merge}\,(\texttt{1}.c)\,(v.s_1)\,(abs.s_2) &= v.\texttt{merge}\,c\,s_1\,s_2 \\
\texttt{merge}\,(\texttt{0}.c)\,(abs.s_1)\,(v.s_2) &= v.\texttt{merge}\,c\,s_1\,s_2
\end{aligned}
$$

Examples

$base = (1)$	1	1	1	1	1	1	1	1	1	1	1	1	$\ldots$
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	$\ldots$
$h = (10)$	1	0	1	0	1	0	1	0	1	0	1	0	$\ldots$
$y = x$ `when` h	x_0		x_2		x_4		x_6		x_8		x_{10}	x_{11}	$\ldots$
$h' = (100)$	1		0		0		1		0		0	1	$\ldots$
$z = y$ `when` h'	x_0						x_6					x_{11}	$\ldots$
k			k_0		k_1				k_2		k_3		$\ldots$
`merge` $h'\ z\ k$	x_0		k_0		k_1		x_6		k_2		k_3		$\ldots$

Sampling and Clocks

- in $x\ \mathtt{when}^{\#}\ y$, x and y must have the same clock cl
- the clock of $x\ \mathtt{when}^{\#}\ c$ is noted $cl\ \mathtt{on}\ c$: it means that c moves at the pace cl

$$
\begin{array}{lcl}
s\ \mathtt{on}\ c & = & \epsilon \text{ if } s = \epsilon \text{ or } c = \epsilon \\
(\mathtt{1}.cl)\ \mathtt{on}\ (\mathtt{1}.c) & = & \mathtt{1}.cl\ \mathtt{on}\ c \\
(\mathtt{1}.cl)\ \mathtt{on}\ (\mathtt{0}.c) & = & \mathtt{0}.cl\ \mathtt{on}\ c \\
(\mathtt{0}.cl)\ \mathtt{on}\ (abs.c) & = & \mathtt{0}.cl\ \mathtt{on}\ c
\end{array}
$$

We get:

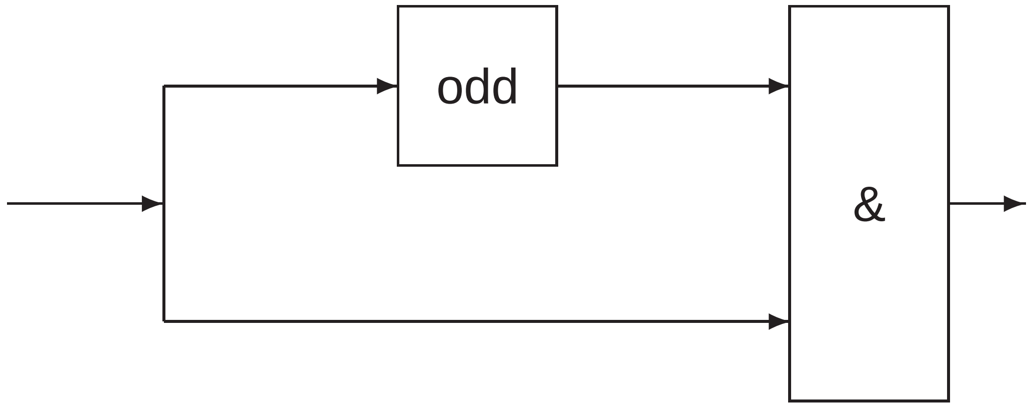
$$\mathtt{when} : \forall cl.\forall x : cl.\forall c : cl.cl\ \mathtt{on}\ c$$

$$\mathtt{merge} : \forall cl.\forall c : cl.\forall x : cl\ \mathtt{on}\ c.\forall y : cl\ \mathtt{on}\ \mathtt{not}\ c.cl$$

Fo any clock cl, if the first input x has clock cl and the second input c has clock cl then $x\ \mathtt{when}\ c$ has clock $cl\ \mathtt{on}\ c$.

Checking Synchrony

The previous programs is now statically rejected by the compiler.



This is essentially a **typing problem**:

```
let odd x = x when half
let non_synchronous x = x & (odd x)
                            ^^^^^^^

This expression has clock 'a on half,
but is used with clock 'a
```

In synchronous languages, we only consider **clock equality**

From pure synchrony to N-synchrony

- The comparison of clocks is limited to clock equality, i.e., “two streams are synchronous or not”
- What about comparing streams which are not exactly synchronous but “not far”?
- How to account for possible “gittering” in the system as found in video applications?
- How to model execution time?

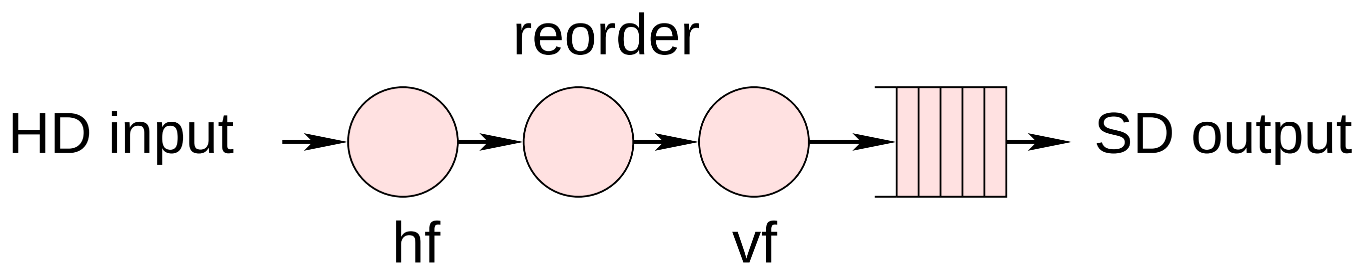
A typical example: the Downscaler

high definition (HD) $\longrightarrow$ 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



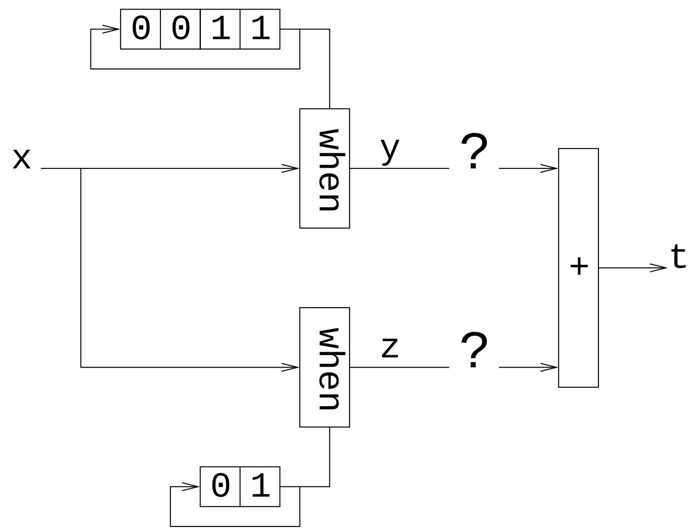
Real-Time Constraints

the input and output processes: 30Hz.

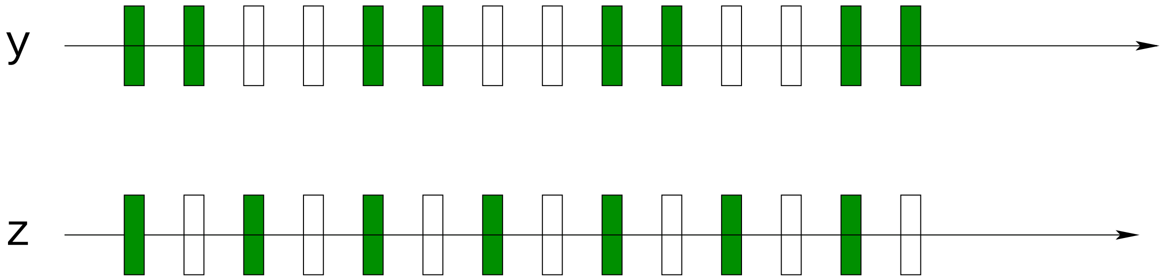
HD pixels arrive at $30 \times 1920 \times 1080 = 62,208,000Hz$

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

But too restrictive for our video applications



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



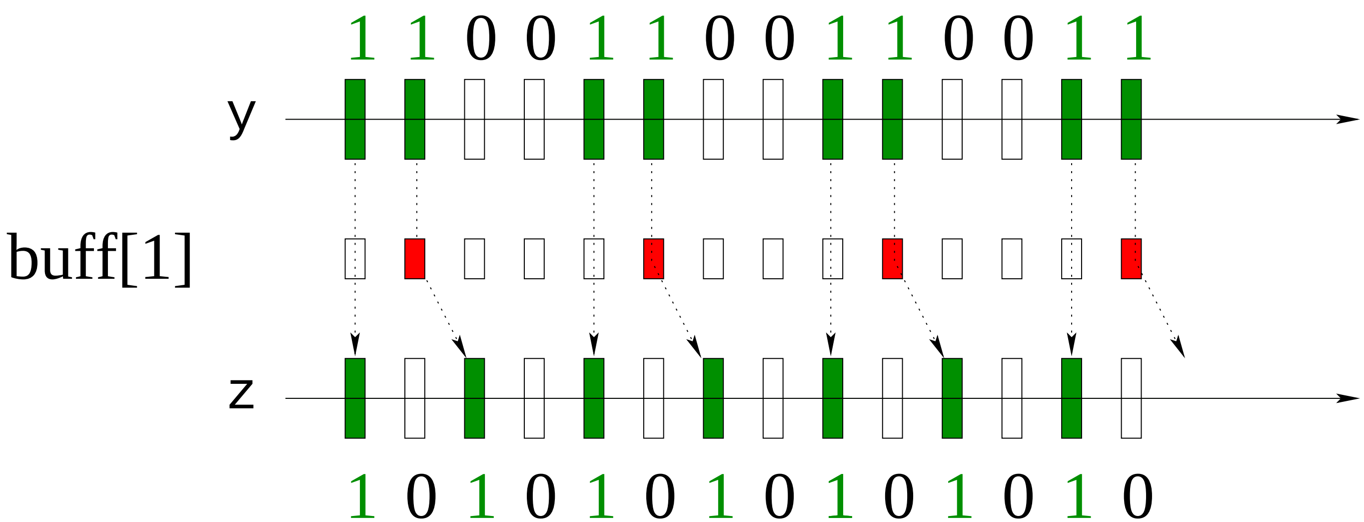
- adding buffer code (by hand) is feasible but hard and error-prone

- 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
- yet compilable to some synchronous code
- allows to compose 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*
- 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

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

- 1 stands for the presence of an event
- 0 for its absence

Definition:

$$w ::= u(v) \quad \text{where } u \in (0+1)^* \text{ and } v \in (0+1)^+$$

Causality order and Synchronisability

Precedence relation: $w_1 \preceq w_2$

- "1s from w_1 arrive before 1s from w_2"
- $\preceq$ is a partial order which abstracts the causality order between streams
- $(\mathbb{Q}_2, \preceq, \sqcup, \sqcap)$ is a lattice

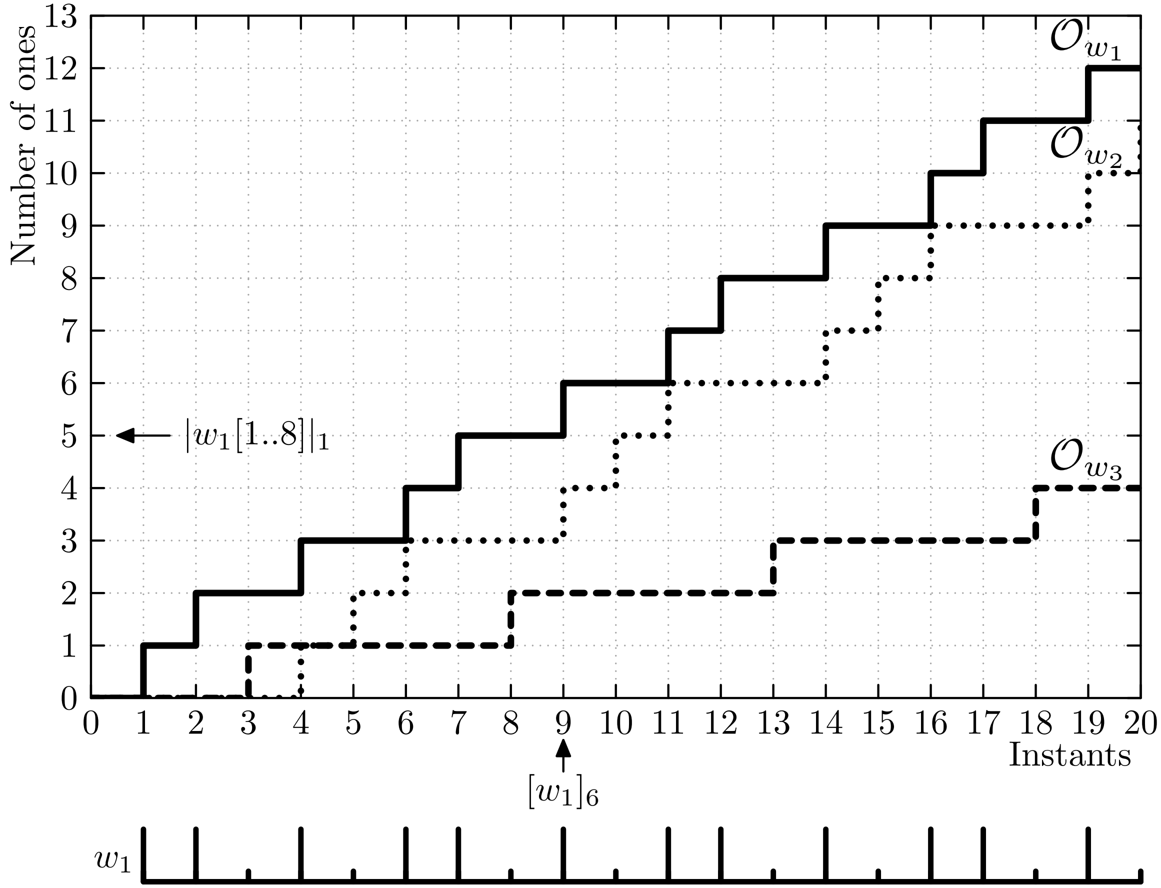
Synchronisability:

Two infinite periodic binary words w and w' are *synchronisable*, noted $w \bowtie w'$ iff it exists $d \in \mathbb{N}$ such that $w \preceq 0^d w'$ and $d' \in \mathbb{N}$ such that $w' \preceq 0^{d'} w$.

1. $11(01)$ and (10) are synchronisable
2. (010) and (10) are not synchronisable since there are too much reads or too much writes (infinite buffers)

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

Clocks represented graphically



Notations:

$w[i]$: element at index i

$[w]_j$: position of the j^{th} 1

$\mathcal{O}_w(i)$: number of $1s$ seen in w until index i.

buffer $size(w_1, w_2) = \max_{i \in \mathbb{N}}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$

precedence $w_1 \preceq w_2 \Leftrightarrow \forall i,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$

synchronizability $w_1 \bowtie w_2 \Leftrightarrow \exists b_1, b_2 \in \mathbb{Z}, \forall i,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$

Multi-sampled Systems (clock sampling)

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

c $\mathtt{on}$ w denotes a *subsampled clock*.

c $\mathtt{on}$ w is the clock obtained in advancing in w at the pace of clock c. E.g., $1(10)$ $\mathtt{on}$ $(01) = (0100)$.

$\mathtt{base}$	1	1	1	1	1	1	1	1	1	1	...	(1)
p_1	1	1	0	1	0	1	0	1	0	1	...	$1(10)$
$\mathtt{base}$ $\mathtt{on}$ p_1	1	1	0	1	0	1	0	1	0	1	...	$1(10)$
p_2	0	1		0		1		0		1	...	(01)
$(\mathtt{base}$ $\mathtt{on}$ $p_1)$ $\mathtt{on}$ p_2	0	1	0	0	0	1	0	0	0	1	...	(0100)

Computing with Periodic Clocks

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

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 $|b| = |b'| = \mathrm{lcm}(|v|, |v'|)$

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

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

From Pure-Synchrony to N-Synchrony

Pure-Synchrony:

- Synchrony can be checked using standard type system
- only need clock equality (and clocks are not necessarily periodic)

$$\frac{H \vdash e_1 : ck \qquad H \vdash e_2 : ck}{H \vdash op(e_1, e_2) : ck}$$

N-Synchrony:

- extend the basic clock calculus with a **sub-typing rule**:

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

- defines the synchronisation points where buffer code should be inserted

Going further: what about non periodic systems?

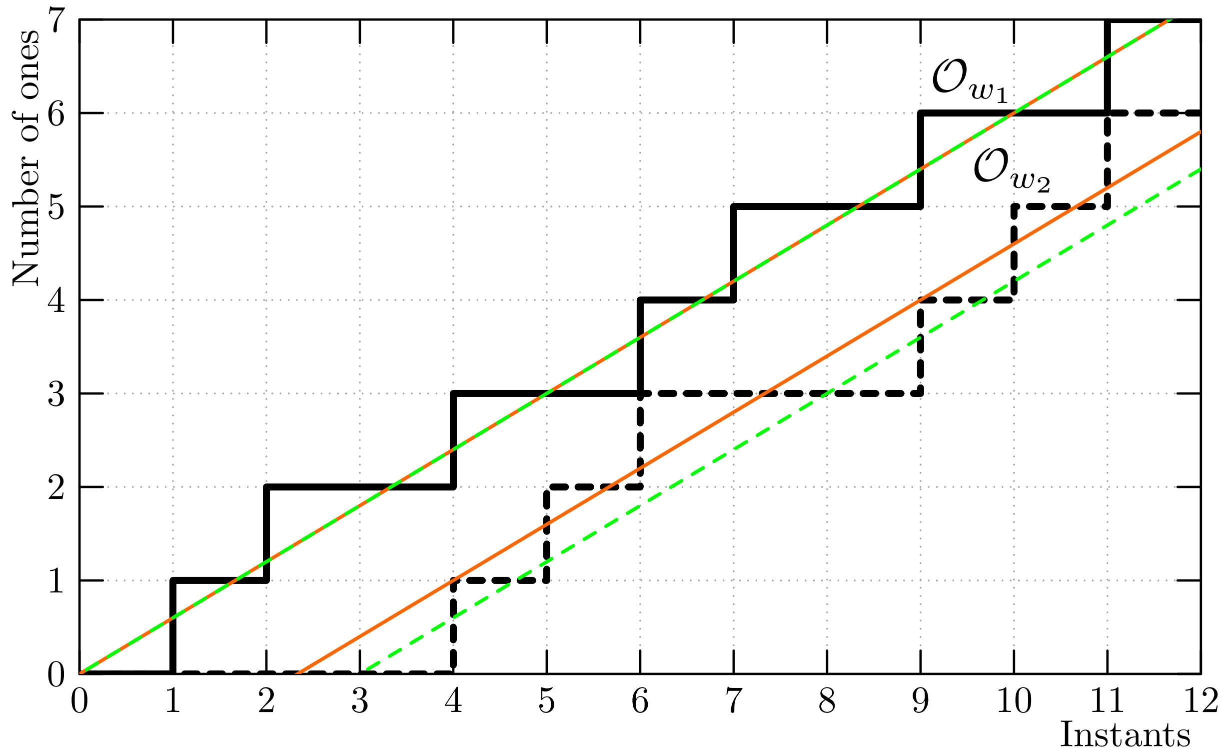
- Introducing clock relations gives more flexibility with as much guaranties as in synchronous model. No deadlocks, no buffer overflows.
- Subtyping relation can be checked provided clocks are periodic.
- Computing with exact period is unfeasible in practice. E.g., (10100100) on $0^{3600}(1)$ on $(101001001) =$ $0^{9600}(10^4 10^7 10^7 10^2)$
- Motivations:
  1. dealing with long patterns in periodic clocks. Avoid exact computation.
  2. specify/model jittering, i.e., how to deal with "almost periodic" clocks ? For instance α on w with $w = 00.(\ (10) + (01)\ )^*$ (e.g. $w = 00\ 01\ 10\ 01\ 01\ 10\ 01\ 10 \ldots$)

Idea: Manipulate sets of clocks instead of clocks. Transform the synchronisation problem into a linear problem with rational numbers.

Clock abstraction (work in progress)

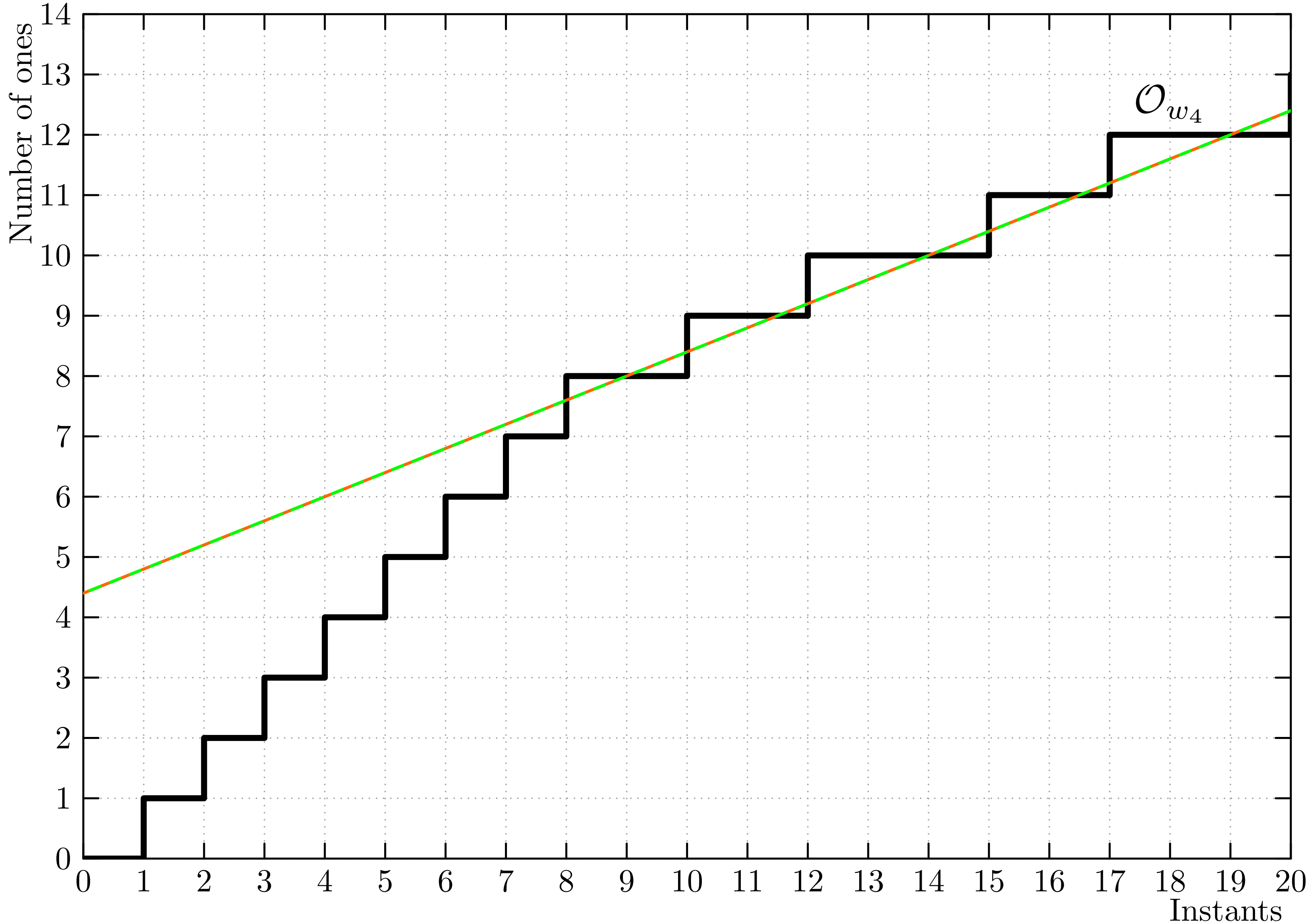
$$concr\left((b^0, b^1, r)\right) \overset{def}{\Leftrightarrow}$$

$$\left\{ w,\ \forall i \geq 1, \quad \begin{array}{rcl} & w[i] = 1 \implies & \mathcal{O}_w(i-1) < r \times i + b^1 \\ \wedge & w[i] = 0 \implies & \mathcal{O}_w(i-1) \geq r \times i + b^0 \end{array} \right\}$$

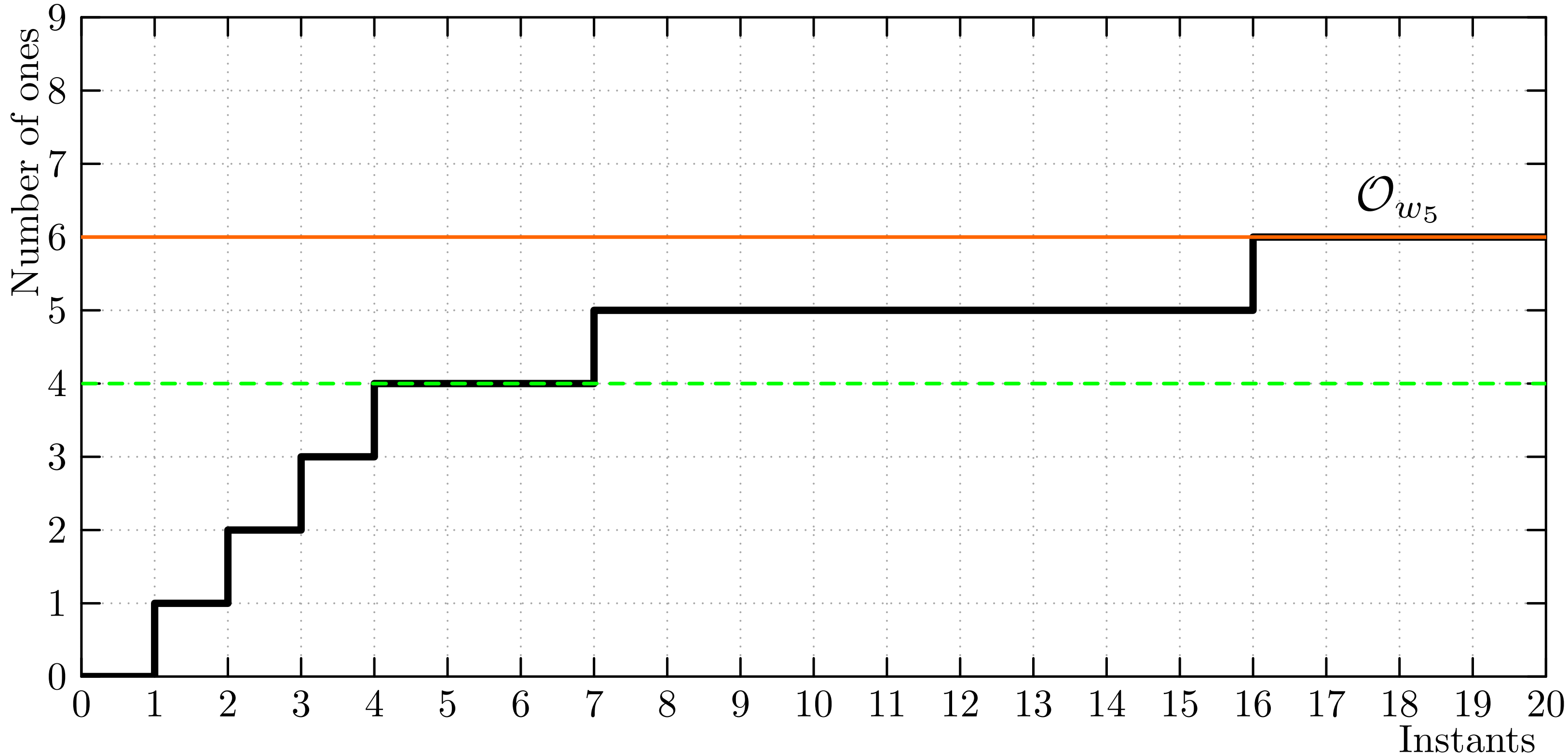


$w_1 : (0, 0, \frac{3}{5})$ and $w_2 : (-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5})$

- Initial sets of $1s$ are well abstracted.



- Clocks with a nul rate can be abstracted.



Properties

Definition 1 ($early_a, late_a$)**.** *Let* $a = \left(b^0, b^1, r\right)$ *be a clock enveloppe.*

$$
\begin{aligned}
early_a &= \sqcap\{w,\ \forall i \geq 1,\ w[i] = 1 \implies \mathcal{O}_w(i-1) < r \times i + b^1\} \\
late_a &= \sqcup\{w,\ \forall i \geq 1,\ w[i] = 0 \implies \mathcal{O}_w(i-1) \geq r \times i + b^0\}
\end{aligned}
$$

Proposition 1 (bounds of the enveloppe)**.**

$\forall w \in concr(a),\ (early_a \preceq w) \wedge (w \preceq late_a).$

Proposition 2 (Empty concretisation)**.** $\forall a,\ concr(a) = \emptyset \Leftrightarrow early_a \npreceq late_a.$

Proposition 3 (Early and Late binary words)**.**

$$
\begin{aligned}
\forall i, \quad \mathcal{O}_{early_a}(i) &= \max(0, \min(i, \lceil r \times i + b^1 \rceil)) \\
\mathcal{O}_{late_a}(i) &= \max(0, \min(i, \lceil r \times i + b^0 \rceil)
\end{aligned}
$$

Proposition 4 (Non-empty enveloppe)**.**

$\forall a = \left(b^0, b^1, r\right),\ b^0 \leq b^1 \implies concr(a) \neq \emptyset.$

Proposition 5 (Perfect Periodic Clock)**.** $\left|concr\left(b^0, b^1, r\right)\right| = 1.$

Clock enveloppes as circuits (i.e., automata)

Given (b_0, b_1, r), write a generator/acceptor of clocks within an enveloppe: this is indeed a synchronous circuit (here written in Lucid Synchrone syntax)

`<:`, `*:`, etc. are the classical operation lifted to rational.

```
type rat = { num: int; den: int }

let norm ({ num = n; den = l }, i, j) =
  if i >= l && j >= n then (i - l, j - n) else (i, j)

let node check((b0, b1, r), clk) = ok where
  rec i, j = (1,0) fby norm(r, i+1, if clk then j + 1 else j)
  and ok = if clk
           then (rat_of_int j) <: r *: (rat_of_int i) +: b1
           else (rat_of_int j) >=: r *: (rat_of_int i) +: b0
```

We only need integer arithmetic. In the same way, we can implement a generator, which either non-deterministically produce a clock within an enveloppe or the *early* or *late* bounds.

Abstract Operators: $\mathit{not}^{\sim}$ and $\mathit{on}^{\sim}$

Replace exact computation with not and on by abstract ones.

$$\mathit{not}^{\sim}\ \left(\left(b^0, b^1, r\right)\right) = \left(-b^1, -b^0, 1 - r\right)$$

Property: $a = \mathit{not}^{\sim}\ \mathit{not}^{\sim}\ a$

If ${b^0}_1 \leq 0$ and ${b^0}_2 \leq 0$:

$$\left({b^0}_1, {b^1}_1, r_1\right)\ \mathit{on}^{\sim}\ \left({b^0}_2, {b^1}_2, r_2\right) = \left({b^0}_{12}, {b^1}_{12}, r_{12}\right)$$

with: $r_{12} = r_1 \times r_2,\ {b^0}_{12} = {b^0}_1 \times r_2 + {b^0}_2,\ {b^1}_{12} = {b^1}_1 \times r_2 + {b^1}_2$

Abstraction of a sampled clock:

We are able to abstract a composed clock without computing the associated binary word.

$$\begin{aligned} abs(\texttt{not}\ w) &\overset{def}{\Leftrightarrow} \mathit{not}^{\sim}\ abs(w) \\ abs(c_1\ \mathit{on}\ c_2) &\overset{def}{\Leftrightarrow} abs(c_1)\ \mathit{on}^{\sim}\ abs(c_2) \end{aligned}$$

Proposition: Those operations are correct, i.e., $c \in concr(abs(c))$

Abstract Relations: $\bowtie^{\sim}$, $\preceq^{\sim}$, $<:^{\sim}$

If the abstract relation is verified, the concrete one is verified on all elements of the respective conretization sets

$$({b^0}_1, {b^1}_1, r_1) \bowtie^{\sim} ({b^0}_2, {b^1}_2, r_2) \Leftrightarrow r_1 = r_2$$

Proposition: $abs(c_1) \bowtie^{\sim} abs(c_2) \Leftrightarrow c_1 \bowtie c_2$

Checking precedence is checking an arithmetic inequality

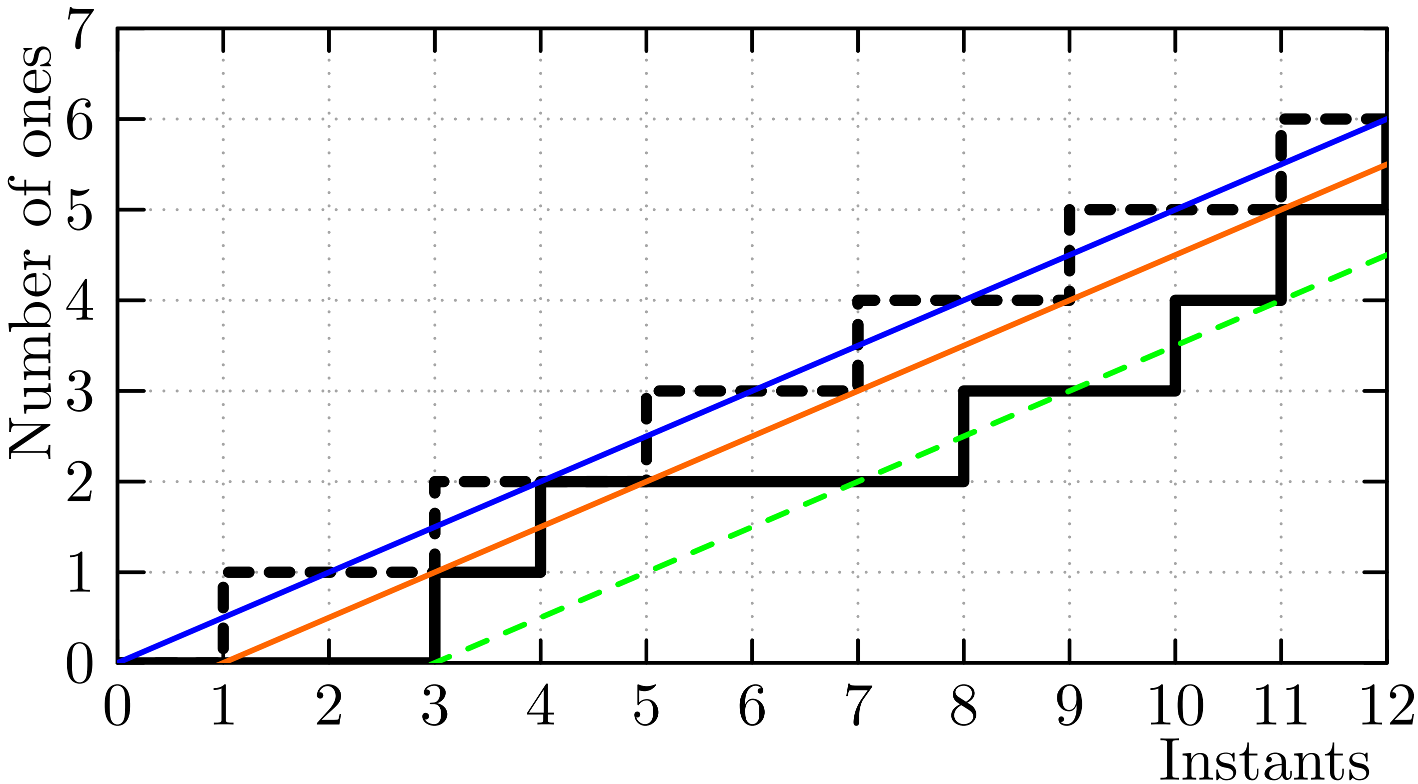
$${b^0}_1 \geq {b^1}_2 \implies a_1 \preceq^{\sim} a_2$$

Proposition: $abs(c_1) \preceq^{\sim} abs(c_2) \Rightarrow c_1 \preceq c_2$

$$a_1 <:^{\sim} a_2 \Leftrightarrow a_1 \bowtie^{\sim} a_2 \wedge a_1 \preceq^{\sim} a_2$$

$\implies$ Subtyping can be checked in constant time.

Modelizing Execution Time



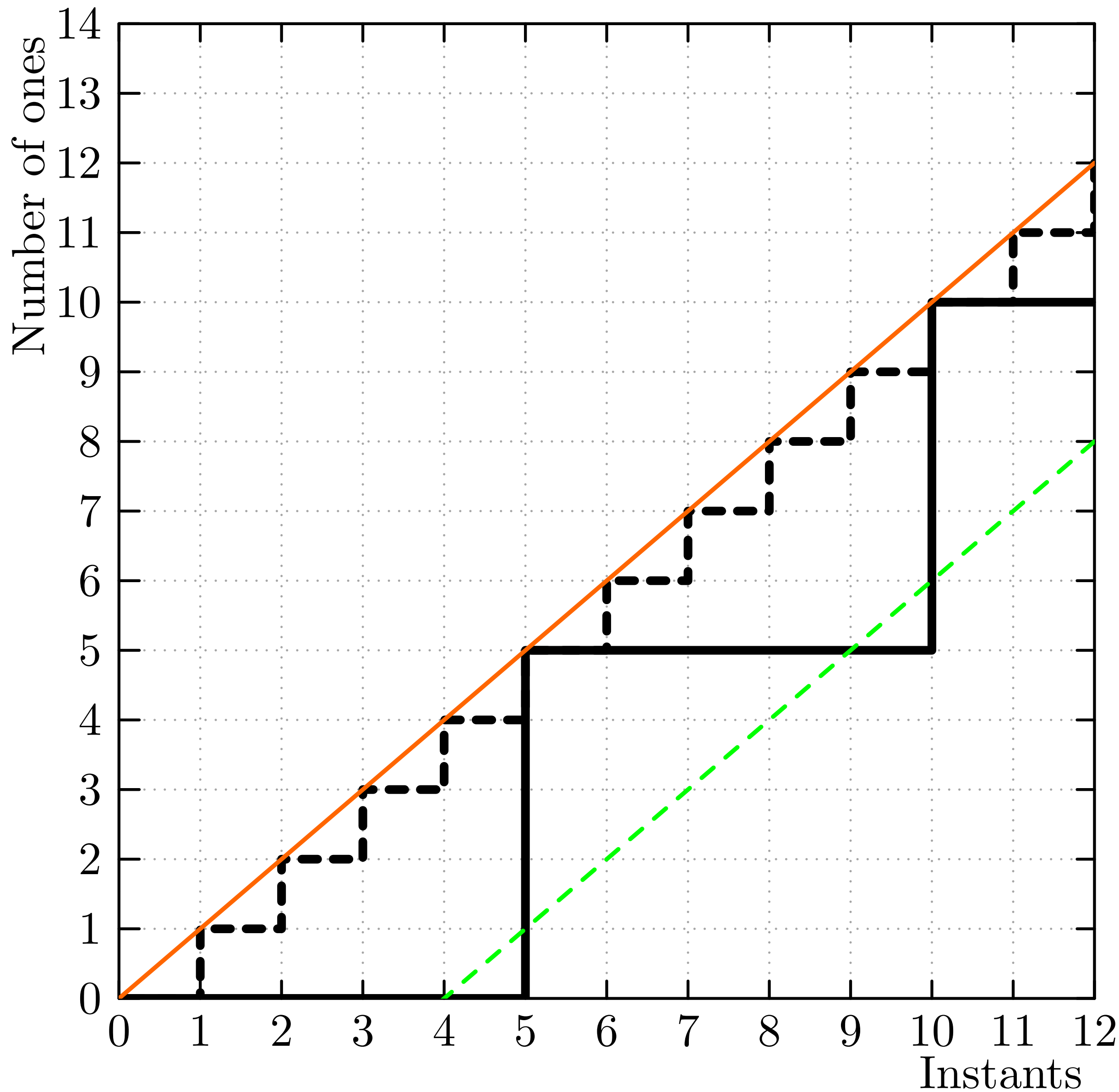
$f :: \forall\alpha.\alpha \; \texttt{on}^{\sim} \; \left(0, 0, \frac{1}{2}\right) \rightarrow \alpha \; \texttt{on}^{\sim} \; \left(-\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}\right)$ i.e.

$\alpha \; \texttt{on}^{\sim} \; \left(0, 0, \frac{1}{2}\right) \rightarrow \alpha \; \texttt{on}^{\sim} \; (-3, -1, 1) \; \textit{on}^{\sim} \; \left(0, 0, \frac{1}{2}\right)$

f must be executed every 2 cycles and that its computation takes between one and three cycles

Composed twice: $f \; o \; f :: \forall\alpha.\alpha \; \texttt{on}^{\sim} \; \left(0, 0, \frac{1}{2}\right) \rightarrow \alpha \; \texttt{on}^{\sim} \; \left(-\frac{6}{2}, -\frac{2}{2}, \frac{1}{2}\right)$

Modelizing Several Reads (or writes) at the Same Instant



Conclusion

- Synchronous data-flow as a sub-set of Kahn Process Networks
- Synchrony means the existence of a common time scale between two communicating processes
- Checking synchrony is mainly a typing problem
- Relaxing synchrony to model a larger class of systems, yet ensuring bounded buffering communication
- algebraic properties on clock sequences (e.g., synchronization, clock enveloppes) have been formalized and proof in the proof assistant Coq (5000 lines)
- We are currently developping a new language to incorporate those clocks

References

[1] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[2] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[3] Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *The Sixth ASIAN Symposium on Programming Languages and Systems (APLAS)*, Bangalore, India, December 2008.

[4] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`.