Clocks in Kahn Process Networks

Marc Pouzet
École normale supérieure

MPRI

Course notes

October 2023

Kahn Process Networks

A system is defined by a set of operators that run in parallel and communicate with FIFO queues.

It can be represented by a set of equations :

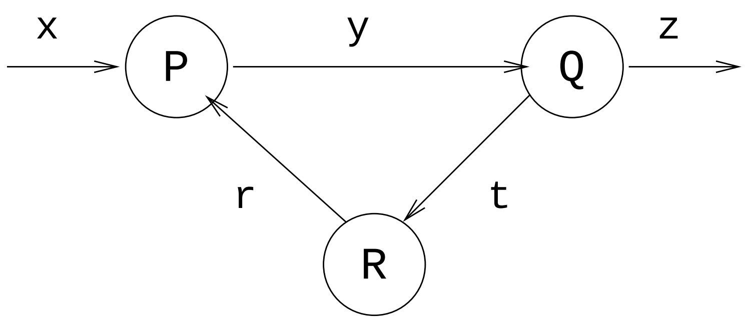
$$\begin{aligned} z, t &= Q(y) \\ y &= P(x, r) \qquad\qquad z = F(x) \\ r &= M(t) \end{aligned}$$

What is the meaning (semantics) of these two sets of equations :
Meaning of z, t, y, P, Q, M ? Knowing them, what is the meaning of F ?

Modularity : the two (left and right) sets of equations should define the same relation between x and z, i.e., naming a set of equations should not change the semantics of the system.

Kahn Networks [Kahn'74, Kahn'75]

In the 70's Kahn showed that the semantics of deterministic parallel processes communicating through (possibly) unbounded buffers is a stream function.



- A set of sequential deterministic processes (i.e., sequential programs) written in an imperative language : `P`, `Q`, `M`,...
- They communicate **asynchronously** via **message passing** into FIFOs (buffers) using two primitives get/put with the following assumptions :
  - Read is blocking on the empty FIFO ; sending is non blocking.
  - Channels are supposed reliable (communication delays are bounded).
  - Read (waiting) on a single channel only, i.e., the program :

    ```
    if (a is not empty) or (b is not empty) then ...
    ```

    is FORBIDDEN

Concretely :

- A buffer channel is defined by two primitives, get, to wait (pop) a value from a buffer and put, to send (push) a value.
- Parallel composition can either be implemented with regular processes ("fork") or lightweight processes ("threads").
- Historically, Gilles Kahn was interested in the semantics of Unix pipes and Unix processes communicating through FIFOs.

E.g., take OCaml :

```
type 'a buff = { put: 'a -> unit; get: unit -> 'a }
val buffer : unit -> 'a buff
```

`buffer ()` creates a buffer associated to a read and write functions.

Either unbounded size (using lists) or statically bounded size. Add a possible status bit : `IsEmptyBuffer` and `IsFullBuffer`.

Implementation

Here is a possible implementation of the previous set of processes (`kahn.ml`).

```
(* Process P *)
let p x r y () =
  y.put 0; (* init *)
  let memo = ref 0 in
  while true do
    let v = x.get () in
    let w = r.get () in
    memo := if v then 0 else !memo + w;
    y.put !memo
  done
(* Process Q *)
let q y t z () =
  while true do
    let v = y.get () in
    t.put v;
    z.put v
  done
```

```
(* Process M *)
let m t r () =
  while true do
    let v = t.get () in
    r.put (v + 1)
  done
```

```
(* Put them in parallel. *)
let main x z () =
  let r = buffer () in let y = buffer () in
  let t = buffer () in
  Thread.create (p x r y) ();
  Thread.create (q y t z) ();
  Thread.create (m t r) ()
```

Question :

— Provide an implementation of the function `buffer` in OCaml (using modules `Thread`, `Mutex`, or `Unix` and `Sys`).

Questions :

— What is the semantics of `p`, `q`, `m` and `main` ?

— What does it change when removing line `(* init *)` ?

— Would you be able to prove that the program `main` is non blocking, i.e, if `x.get ()` never blocks then `z.put ()` never blocks ?

— Is there a statically computable bound for the size of buffers without leading to blocking ?

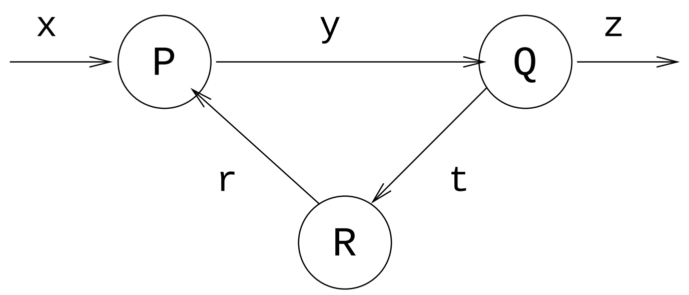
— Would it be possible to statically schedule this set of processes, that is, to generate an equivalent sequential program ?

These are all undecidable questions in the general case (see [Thomas Park's PhD. thesis, 1995], among others).

Kahn Process Networks and variants have been (and are still) very popular, both practically, and theoretically as it conciliates **parallelism** and **determinacy**.

Kahn Process Networks

Kahn Principle : The semantics of process networks communicating through unbounded FIFOs (e.g., Unix pipe, sockets) ?



— message communication into FIFOs (`send`/`wait`)

— reliable channels, bounded communication delay

— blocking wait on a channel. The following program is **forbidden**

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

— a process = a continuous function $(V^\infty)^n \rightarrow (V'^\infty)^m$.

Lustre :

— Lustre has a **Kahn semantics** (no test of absence)

— A dedicated **type system** (clock calculus) to guaranty the existence of an execution with no buffer (no synchronization)

Kahn Process Networks

(+) : Simple semantics : a process defines a function (determinism) ; composition is function composition

(+) : Modularity : a network is a continuous function

(+) : Asynchronous distributed execution : easy ; no centralized scheduler

(+/-) : Time invariance : no explicit timing ; but impossible to state that two events happen at the same time.

(-) : Ressources : KPN can run with unbounded memory or have deadlock. The parallel composition of two bounded memory/deadlock free KPN may be unbounded/deadlock.

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	$\ldots$

Kahn Process Networks

Some restrictions must be imposed for KPN to be used for real-time/embedded applications where ressources (time and memory) matter.

KPNs were quite successful to model for video apps (TV boxes) : Sally (Philips NatLabs), StreamIt (MIT), Xstream (ST-micro) with various "synchronous" restriction *à la SDF* (Edward Lee)

More generally, we are interested here in stream programming :

— A stream models a time evolving value ;

— The time line is the set of natural numbers ;

— A system is a stream function, that is, a function from streams to streams.

A small dataflow kernel

Expression (e), constants (i), functions applied pointwise ($op(e_1, ..., e_n)$), data-flow primitives.

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

Definition of stream functions, equations :

$$\begin{array}{lcl} d & ::= & \texttt{node}\ f(p) = p\ \texttt{with}\ D \\ p & ::= & x, ..., x \mid x \\ D & ::= & x = e \mid D\ \texttt{and}\ D \mid \texttt{var}\ x\ \texttt{in}\ D \end{array}$$

Dataflow 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
$x' = x$ `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_2

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

— If V is a set, V^n is the set of sequences of length n made by concatenating elements from V. $V^\star = \cup_{n=0}^{\infty} V^n$ is the Kleene star operation.

— $V^\infty = V^* \cup V^\omega$ is the set of finite and infinite sequences.

— ϵ is the empty sequence.

— $v.s$ is a sequence whose first element is v and tail is s.

— The set $(V^\infty, \leq, \epsilon)$, with $\leq$ the prefix order between sequences, ϵ the minimum element, is a complete partial order (cpo). [a]

— The Kleene theorem applies : if $f : V^\infty \to V^\infty$ is a continuous function, the equation $x = f(x)$ has a least fix-point $x^\infty = lim_{n\to\infty}(f^n(\epsilon))$.

Every operator is interpreted as a stream. If $x \mapsto s_1$ and $y \mapsto s_2$ then the value of $x + y$ is $lift^2(+)(s_1, s_2)$

a. The minimal element is usually written $\perp$.

Kahn Semantics

$$\begin{array}{lcl}
\mathit{lift}^0(v) & = & v.\mathit{lift}^0(v) \\
\mathit{lift}^1(\mathit{op})(v.s) & = & \mathit{op}(v).\mathit{lift}^1(\mathit{op})(s) \\
\mathit{lift}^1(\mathit{op})(\epsilon) & = & \epsilon \\
\mathit{lift}^2(\mathit{op})(v_1.s_1, v_2.s_2) & = & \mathit{op}(v_1, v_2).\mathit{lift}^2(\mathit{op})(s_1, s_2) \\
\mathit{lift}^2(\mathit{op})(s_1, s_2) & = & \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\mathit{fby}(s_1)(s_2) & = & \epsilon \text{ if } s_1 = \epsilon \\
\mathit{fby}(v_1.s_1)(s_2) & = & v_1.s_2
\end{array}$$

Kahn Semantics

$$
\begin{array}{lcl}
when(v.s, \mathsf{1}.c) & = & v.when(s, c) \\
when(v.s, \mathsf{0}.c) & = & when(s, c) \\
when(s_1, s_2) & = & \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
merge(\mathsf{1}.c, v.s_1, s_2) & = & v.merge(c, s_1, s_2) \\
merge(\mathsf{0}.c, s_1, v.s_2) & = & v.merge(c, s_1, s_2) \\
merge(\mathsf{1}.c, \epsilon, s_2) & = & \epsilon \\
merge(\mathsf{0}.c, s_1, \epsilon) & = & \epsilon \\
merge(\epsilon, s_1, s_2) & = & \epsilon
\end{array}
$$

All those functions are continuous [2].

An other formulation - sequences

Represent a sequence as a function from an initial segment of $\mathbb{N}$ to V.

Initial segment : $I \subseteq \mathbb{N}$ is an initial segment when :

$$\forall n, m \in \mathbb{N}.(n \in I) \wedge (m \leq n) \Rightarrow (m \in I)$$

E.g., $\emptyset$, $\{0, 1, 2\}$ are initial segment ; $\{0, 42\}$ is not.

Lemma : For any subset A of $\mathbb{N}$, there exists a strictly increasing, one-to-one function ϕ_A between an initial segment I_A of $\mathbb{N}$ and A.

A signal u is a sequence $(u_n)_{n \in N}$, finite or not, indexed on an initial segment N.

$$\begin{array}{lcllr}
\mathit{lift}^0(v) & = & (u)_{n\in\mathbb{N}} & \mathit{with} & \forall n \in \mathbb{N}. u_n = v \\
\mathit{lift}^1(op)((u_n)_{n\in N}) & = & (v_n)_{n\in N} & \mathit{with} & \forall n \in N. v_n = op(v_n) \\
\mathit{lift}^2(op)((u_n)_{n\in N}, (v_n)_{n\in N}) & = & (w_n)_{n\in N} & \mathit{with} & \forall n \in N. w_n = op(u_n, v_n) \\
\mathit{fby}((u_n)_{n\in N})((v_n)_{n\in N}) & = & (w_n)_{n\in N} & \mathit{with} & w_0 = u_0 \\
 & & & \mathit{and} & \forall n \in N\backslash\{0\}. w_n = v_{n-1}
\end{array}$$

If $(h_n)_{n\in N}$ is a boolean sequence, define :

$$N_h = \{k \in N \mid h_k = 1\}$$

and

$$N_{\overline{h}} = \{k \in N \mid h_k = 0\}$$

N_h with $N_{\overline{h}}$ form a partition of N.

$$
\begin{array}{lclll}
when((u_n)_{n \in N}, (h_n)_{n \in N}) & = & (v_n)_{n \in I_{N_h}} & with & v_n = u_{\phi_{N_h}(n)} \\
merge((h_n)_{n \in N}, (u_n)_{n \in I_{N_h}}, (v_n)_{n \in I_{N_{\overline{h}}}}) & = & (w_n)_{n \in N} & with & w_n = u_n if\ n \in N_h \\
 & & & and & w_n = v_n if\ n \in N_{\overline{h}}
\end{array}
$$

The base clock is the constant sequence $base$ such that $\forall n \in \mathbb{N}, base_n = 1$.

An encoding in Haskell

```
constant v n = v

lift1 op x n = op(x(n))
notl x = lift1 not

lift2 op x y n = op (x(n)) (y(n))
lift3 op x y z n = op (x(n)) (y(n)) (z(n))

x fby y 0 = x(0)
x fby y n = y(n-1)

x when h n = x(I(h)(n+1))
merge h x y n = if h(n) then x(O(h)(n)) else y(O(notl h)(n)
```

where `I` and `O` are (respectively) the index and cumulative functions.

The index and cumulative functions

I and O functions If h is a boolean stream, $O(h)(n)$ is the sum of 1 till index n ; $I(h)(n)$ is the index of the n-th 1 in h.

$$O(h)(n) = \sum_{i=0}^{n} h(i) \quad I(h)(n) = min\{k \in \mathbb{N} \mid O_h(k) = n\}$$

```
O(h)(n) = h(n) + (if n = 0 then 0 else O(h)(n-1))

I(h)(n) = I'(h)(0)(n)
I'(h)(i)(n) = if h(i) then if n = 1 then i
                           else I'(h)(i+1)(n-1)
                else I'(h)(i+1)(n)
```

It is very possible that $I(n)$ be undefined (no value in $\mathbb{N}$). E.g., `(x when (constant false))(n)`. The domain of a signal x (values of n for which $x(n)$ exists) is an initial section.

An encoding of sequences in Ocaml (see companion file)

```
type 'a sequence = int -> 'a


let const v n = v
let extend f x n = (f(n)) (x(n))
let lift1 f x = extend (const f) x
let lift2 f x y = extend (extend (const f) x) y
let lift3 f x y z = extend (extend (extend (const f) x) y) z


let notl x = lift1 (fun x -> not x) x
let plusl x y = lift2 (+) x y
let minusl x y = lift2 (-) x y
let andl x y = lift2 (&&) x y
let eql x y = lift2 (=) x y
(* The [if/then/else] of Lustre *)
let mux x y z = lift3 (fun x y z -> if x then y else z) x y z
```

Cont. (unit delay)

```
let pre v x n = if n = 0 then v else x(n-1)
let fby x y n = if n = 0 then x 0 else y(n-1)

let next x n = x (n+1)
```

Cont. (index/cumulative functions)

```
(* cumulative and index functions *)
(* [cumul(h)(n)] returns the sum of ones in h up to index n *)
let rec cumul(h)(n) = (if h(n) then 1 else 0) +
                      (if n = 0 then 0 else cumul(h)(n-1))
(* [index(h)(n)] returns the index of the n-th one in h *)
let rec index(h)(n) = index_aux(h)(0)(n)
and index_aux(h)(i)(n) =
  if h(i) then if n = 1 then i else index_aux(h)(i+1)(n-1)
  else index_aux(h)(i+1)(n)

(* reset. [reset(h)(n)] returns the greatest *)
(* index i in [0..n] where h(i) = true; if there is no, returns 0 *)
let reset(h)(n) =
  let rec o n = if n = 0 then 0 else if h n then n else o(n-1) in
  o(n)
```

Cont. (filtering/merging - when/merge)

```
(* filtering and merge *)

let whenc x h n = x(index(h)(n+1))

let when_notc x h n = x(index(notl h)(n+1))

let merge h x y n = if h(n) then x(cumul(h)(n)) else y(cumul(notl h)(n))



(* Fixpoint operator over sequences *)

let fix : ('a sequence -> 'a sequence) -> 'a sequence =

  fun f n -> let rec o n = f o n in o n


(* the bottom element [eps ()] *)

let rec eps n = eps n
```

Cont. (examples)

```
(* Examples *)
let half () = let rec half n = pre false (notl half) n in half
let from v =
  let rec f n = pre 0 (plusl f (const 1)) n in f

let incr v x n = pre v (plusl x (const 1)) n
let from v = fix (incr (v(0)))

let sum x =
  (* [The concrete syntax in Zelus is: let rec y = (0 fby y) + x in y] *)
  fix (fun y -> plusl (pre 0 y) x)
```

Cont. examples

```
(* Deadlock / infinite loop *)
(* to test, type [deadlock 42] *)
let id x n = x n
let deadlock n = fix id n

(* to test, type [deadlock 42] *)
let deadlock n =
  let x = const true in
  whenc x (const false) n

(* non synchronous *)
let unbounded =
  let x = const true in
  let h = half () in
  andl (whenc x h) x
```

Cont. examples

```
(* semantics of a simple language with a control structure and a reset *)

(* model a conditional that activate one block when a condition is true;
 *- activate the other when the condition is false *)
let cond_act c f g x =
  (* cond_act c f g x = merge c (f (x when c)) (g (x whenot c))) *)
  fun n -> merge c (f (whenc x c)) (g (when_notc x c))

(* after. [after x k] returns the sub-sequence of x that starts at index k *)
let after x k = fun n -> x (n+k)

(* model a modular reset.
*- take a system [f] and input [x];
*- run f x; every time [c] is true, restart [f] with the remaining inputs *)
let reset_act f x c = fun n -> let k = reset c n in f (after x k)(n-k)
```

Comments

A denotational semantics and reference interpreter in a few lines of OCaml (or any general purpose language).

Yet, it is totally useless in practice.

Think of the (exponential) time (and memory) to compute Fibonacci :

```
let rec fibo(n) = if n <= 1 then 1 else fibo(n-1) + fibo(n-2)
```

whereas it denotes a simple stream equation (written in ZÉLUS) which can be implemented with two registers and run in constant time.

```
let rec pfibo = 1 fby (fibo + pfibo)
and fibo = 1 fby pfibo in fibo
```

Rmq : (1) it cannot be defined in Coq (unless using a trick, e.g., “fuel”) ; (2) yet, it can be used for a relational semantics ; anwering questions like (3) is this program deadlocks ? (4) is this program be implemented in bounded time and space ? are difficult.

An Implementation in Haskell using a Lazy Data-structure

```
module Streams where

data ST a = Cons a (ST a) deriving Show
-- lifting constants
constant x = Cons x (constant x)

-- pointwise application
extend (Cons f fs) (Cons x xs) = Cons (f x) (extend fs xs)

-- delays
(Cons x xs) `fby` y = Cons x y
pre x y = Cons x y

-- sampling
(Cons x xs) `when` (Cons True cs)  = (Cons x (xs `when` cs))
(Cons x xs) `when` (Cons False cs) = xs `when` cs

merge (Cons True c)  (Cons x xs) y  = Cons x (merge c xs y)
merge (Cons False c) x (Cons y ys)  = Cons y (merge c x ys)
```

An embedding in Haskell

function definition/applications are the regular ones ; mutually recursive definitions of streams are represented as mutually recursive definitions of values.

We can write many usefull examples and benefit from features of the host language.

```
lift2 f x y = extend (extend (constant f) x) y
plusl x y = lift2 (+) x y

-- integers greaters than n
from n =
  let nat = n ‘fby‘ (plusl nat (constant 1)) in nat

-- resetable counter
reset_counter res input =
  let output = ifthenelse res (constant 0) v
      v = ifthenelse input
                     (pre 0 (plusl output (constant 1)))
                     (pre 0 output)
  in output
```

Multi-periodic systems

```
every n =
  let o = reset_counter (pre 0 o = n - 1)
                        (constant True)
  in o

filter n top = top when (every n)

hour_minute_second top =
  let second = filter (constant 10) top in
  let minute = filter (constant 60) second in
  let hour = filter (constant 60) minute in
  hour,minute,second
```

Over-sampling (with fixed step)

Compute the sequence $(o_n)_{n\in I\!N}$ such that $o_{2n} = x_n$ and $o_{2n+1} = x_n$.

```
-- the half clock
half = (constant True) `fby` notl half


-- double its input
stutter x =
  o = merge half x ((pre 0 o) when notl half) in o
```

- over-sampling : the internal rate is faster than the rate of inputs
- this is still a real-time program
- why is it rejected in LUSTRE ?

Over-sampling with variable step

Compute the root of an input x (using Newton method)

$$u_n = u_{n-1}/2 + x/2u_{n-1}$$

$$u_1 = x$$

```
eps = constant 0.001
root input =
  let ic = merge ok input (pre 0 ic) when notl ok)
      uc = (pre 0 uc) / 2 + (ic / 2 * pre 0 uc)
      ok = true -> uc - pre 0 uc <= eps
      output = uc when ok
  in output
```

This example mimics an internal while loop (example due to Paul Le Guernic)

Where are the monsters ?

A stream is represented as a lazy data-structure. Nonetheless, lazyness allows streams to be build in a strange manner.

Structural (Scott) order :

$\perp \leq_{struct} v$, $(v : w) \leq_{struct} (v' : w')$ iff $v \leq_{struct} v'$ and $w \leq_{struct} w'$.

The following programs are perfectly correct in Haskell (with a unique non-empty solution)

```
hd (x:y) = x
tl (x:y) = y
incr (x:y) = (x+1) : incr y

one = 1 : one
x = (if hd(tl(tl(tl(x)))) = 5 then 3 else 4) : 1 : 2 : 3 : one
output = (hd(tl(tl(tl(x))))) : (hd(tl(tl(x)))) : (hd(x)) : output
```

The values are :

— $x = 4 : 1 : 2 : 3 : 1 : ...$

— $output = 3 : 2 : 4 : 3 : 2 : 4 : ...$

These stream may be constructed lazilly :

- $x^0 = \bot, x^1 = \bot : 1 : 2 : 3 : un, x^2 = 4 : 1 : 2 : 3 : one.$
- $output^0 = \bot, output^1 = 3 : 2 : 4 : ...$

An other example (due to Paul Caspi) :

```
nat = zero ‘fby‘ (incr nat)
ifn n x y = if n <= 9 then hd(x) : ifn (n+1) (tl(x)) (tl(y)) else y
if9 x y = ifn 0 x y


x = if9 (incr (tl x)) nat
```

We have $x = 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 11, 12, 13, 14, 15, ...$

Are they reasonnable programs ? Streams are constructed in a reverse manner from the future to the past. We say that they are not “causal”.

This is because the structural order between streams allows to fill the holes in any order, e.g. :

$$(\bot : \bot) \leq (\bot : \bot : \bot : \bot) \leq (\bot : \bot : 2 : \bot) \leq (\bot : 1 : 2 : \bot) \leq (0 : 1 : 2 : \bot)$$

It is also possible to build streams with intermediate holes (undefined values in the middle) through the final program is correct :

$$half = 0.\bot.0.\bot...$$

```
fail = fail
half = 0:fail:half
fill x = (hd(x)) : fill (tl(tl x))
ok = fill half
```

We need to model **causality**, that is, stream should be produced in a sequential order. We take the **prefix order** introduced by Kahn :

Prefix order :

$x \leq y$ if x is a prefix of y, that is : $\bot \leq x$ and $v.x \leq v.y$ if $x \leq y$

Causal function :

A function is causal when it is monotonous for the prefix order :

$$x \leq y \Rightarrow f(x) \leq f(y)$$

All the previous program will get the value $\bot$ in the Kahn semantics.

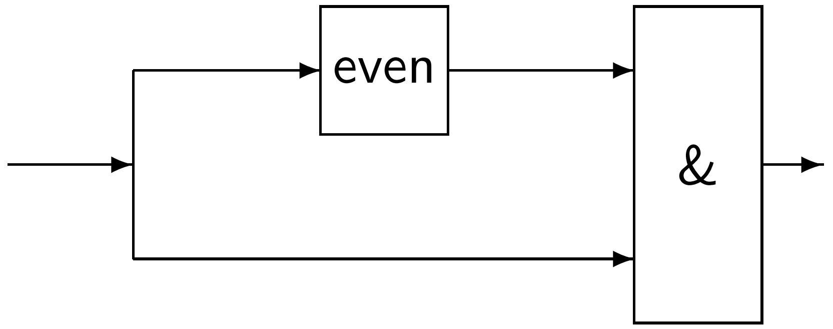
Kahn Semantics in Haskell

It is possible to remove possible non causal streams by forbidding values of the form $\perp.x$. In Haskell, the annotation `!a` states that the value with type a is strict ($\neq \perp$).

```
module SStreams where
-- only consider streams where the head is always a value (not bot)
data ST a = Cons !a (ST a) deriving Show
constant x = Cons x (constant x)

extend (Cons f fs) (Cons x xs) = Cons (f x) (extend fs xs)

(Cons x xs) `fby` y = Cons x y

(Cons x xs) `when` (Cons True cs)  = (Cons x (xs `when` cs))
(Cons x xs) `when` (Cons False cs) = xs `when` cs

merge (Cons True c)  (Cons x xs) y  = Cons x (merge c xs y)
merge (Cons False c) x (Cons y ys)  = Cons y (merge c x ys)
```

This time, all the previous non causal programs have value $\perp$ (stack overflow).

Some "synchrony" monsters



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

Unbounded FIFOs !

- must be rejected statically
- every operator is finite memory through the composition is not : all the complexity (synchronization) is hidden in communication channels
- the Kahn semantics does not model time, i.e., impossible to state that two event arrive **at the same time**

Synchronous (Clocked) streams

Complete streams with an explicit representation of absence (abs).

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

Clock : the clock of x is a boolean sequence

$$\begin{aligned}
&\mathbb{B} = \{\mathtt{0}, \mathtt{1}\}\\
&\mathcal{CLOCK} = \mathbb{B}^{\infty}\\
&\mathtt{clock}\ \epsilon && = && \epsilon\\
&\mathtt{clock}\ (abs.x) && = && \mathtt{0}.\mathtt{clock}\ x\\
&\mathtt{clock}\ (v.x) && = && \mathtt{1}.\mathtt{clock}\ x
\end{aligned}$$

Synchronous streams :

$$ClStream(V, cl) = \{s/s \in (V^{abs})^{\infty} \wedge \mathtt{clock}\ s \leq_{prefix} cl\}$$

An other possible encoding : $x : (V \times \mathbb{N})^{\infty}$

Dataflow Primitives

Constant :

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

Point-wise application :

Synchronous arguments must be constant, i.e., having the same clock

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

Partial definitions

What happens when one element is present and the other is absent ?

Constraint their domain :

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

i.e., $(+)$ expect its two input stream to be on the same clock cl and produce an output on the same clock

These extra conditions are **types** which must be statically verified

Remark (notation) : Regular types and clock types can be written 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 following, we only consider the clock type.

Sampling

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

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

Examples

$base = (1)$	1	1	1	1	1	1	1	1	1	1	1	1	...
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}	...
$h = (10)$	1	0	1	0	1	0	1	0	1	0	1	0	...
$y = x$ `when` h	x_0		x_2		x_4		x_6		x_8		x_{10}	x_{11}	...
$h' = (100)$	1		0		0		1		0		0	1	...
$z = y$ `when` h'	x_0						x_6					x_{11}	...
k			k_0		k_1				k_2		k_3		...
`merge` h' z k	x_0		k_0		k_1		x_6		k_2		k_3		...

```
let clock five =
  let rec f = true fby false fby false fby false fby f in f
let node stutter x = o where
  rec o = merge five x ((0 fby o) whenot five) in o
```

`stutter`$(nat) = 0.0.0.0.1.1.1.1.2.2.2.2.3.3...$

Sampling and clocks

- $x\,\mathtt{when}^{\#}\,y$ is defined when x and y have the same clock cl
- the clock of $x\,\mathtt{when}^{\#}\,c$ is written $cl\ \mathsf{on}\ c$: "c moves at the pace of cl"

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

We get :

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

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

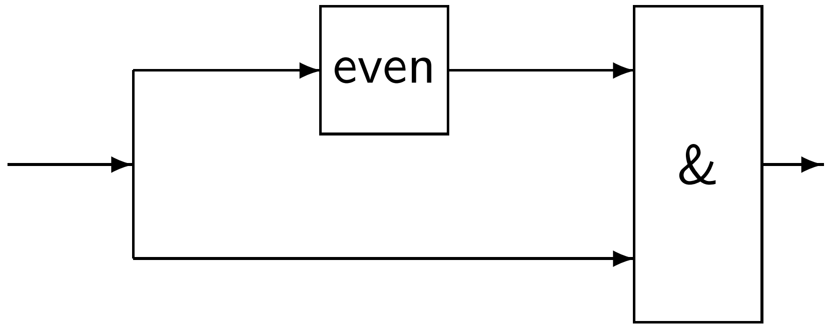
Written instead :

$$\mathtt{when} : \forall cl.cl \rightarrow (c : cl) \rightarrow cl\ \mathsf{on}\ c$$

$$\mathtt{merge} : \forall cl.(c : cl) \rightarrow cl\ \mathsf{on}\ c \rightarrow cl\ \mathsf{on}\ \mathit{not}\ c \rightarrow cl$$

Checking Synchrony

The previous program is now rejected.



This is a now a **typing error**

```
let even x = x when half
let non_synchronous x = x & (even x)
                            ^^^^^^^^
This expression has clock 'a on half,
but is used with clock 'a
```

Final remarks :

— We only considered **clock equality**, i.e., "two streams are either synchronous or not"

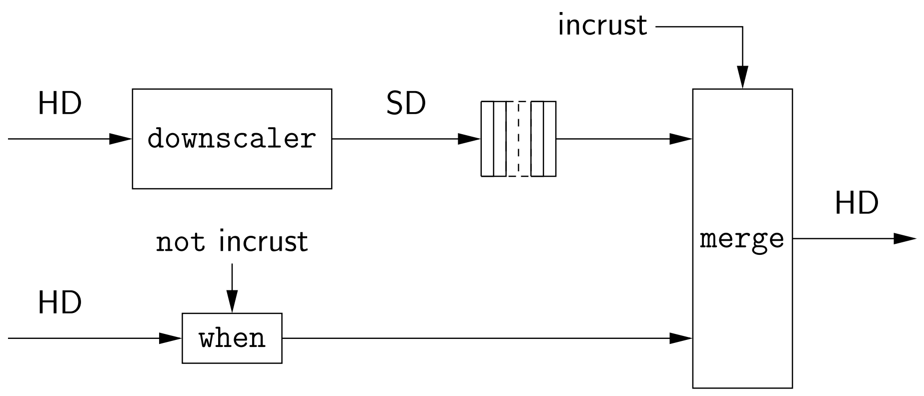
— Clocks are used extensively to generate **efficient sequential code**

From Synchrony to Relaxed Synchrony [a]

- can we compose non strictly synchronous streams provided their clocks are closed from each other ?
- communication between systems which are “almost” synchronous
- model jittering, bounded delays
- Give more freedom to the compiler, generate more efficient code, translate into regular synchronous code if necessary

a. Joint work with Albert Cohen, Marc Duranton, Louis Mandel and Florence Plateau (PhD. Thesis at `https://www.lri.fr/~mandel/lucy-n/~plateau/`)

A typical example : Picture in Picture



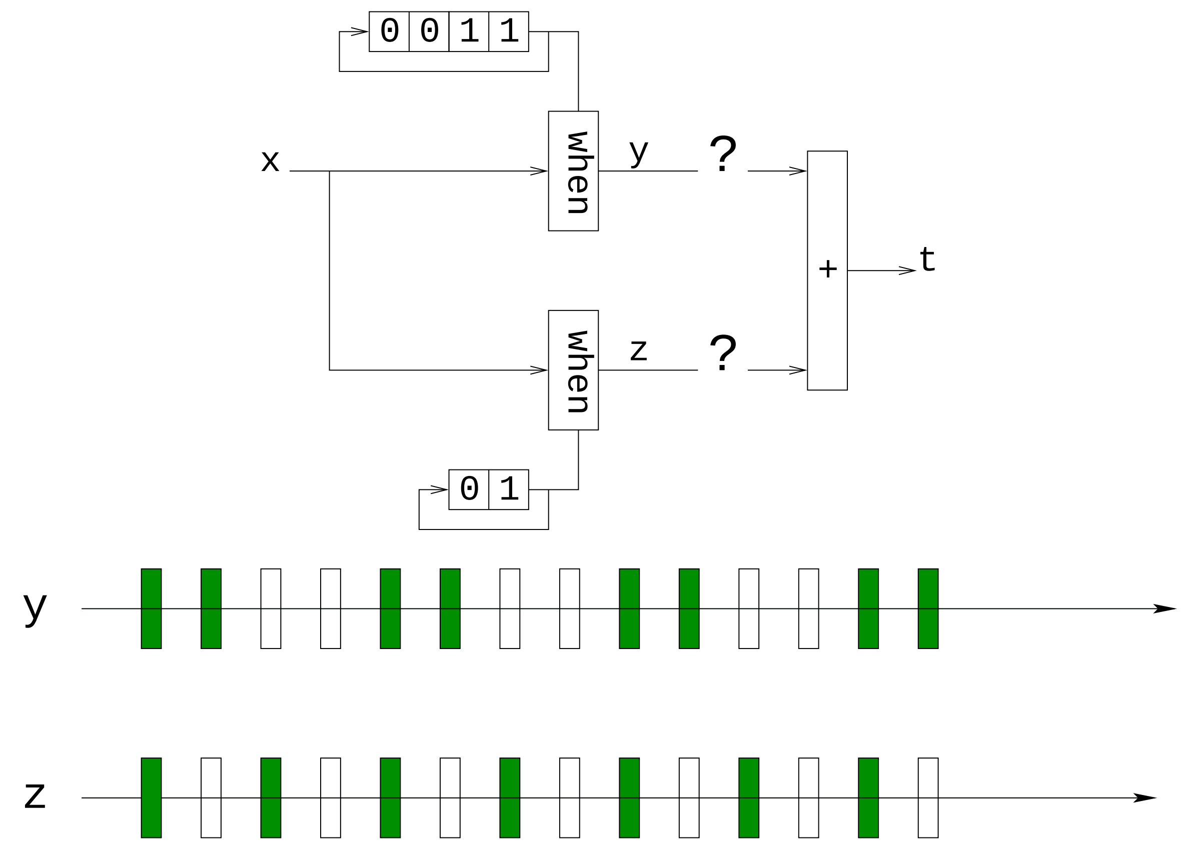
Incrustation of a Standard Definition (SD) image in a High Definition (HD) one

- `downscaler` : reduction of an HD image (1920×1080 pixels) to an SD image (720×480 pixels)
- `when` : removal of a part of an HD image
- `merge` : incrustation of an SD image in an HD image

Question :

- buffer size needed between the `downscaler` and the `merge` nodes ?
- delay introduced by the picture in picture in the video processing chain ?

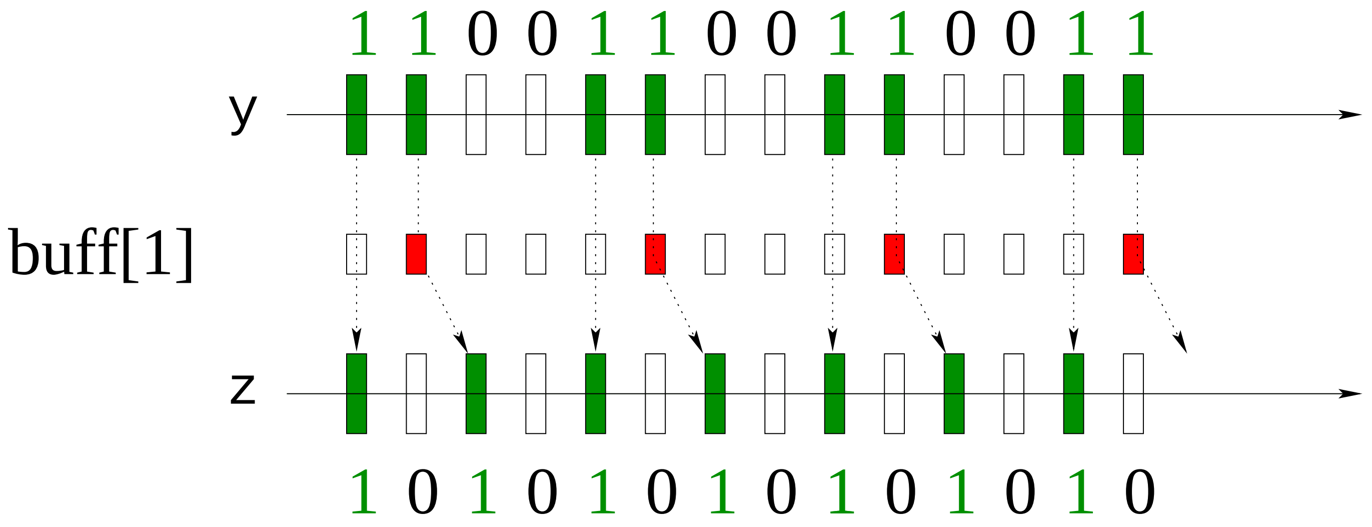
Too restrictive for video applications



- streams should be synchronous
- adding buffer (by hand) difficult and error-prone
- compute it automatically and generate synchronous code

relax the associated clocking rules

N-Synchronous Kahn Networks



- based on the use of *infinite ultimately periodic sequences*
- a precedence relation $cl_1 <: cl_2$

Ultimately periodic sequences

$\mathbb{Q}_2$ for the set of infinite periodic binary words.

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

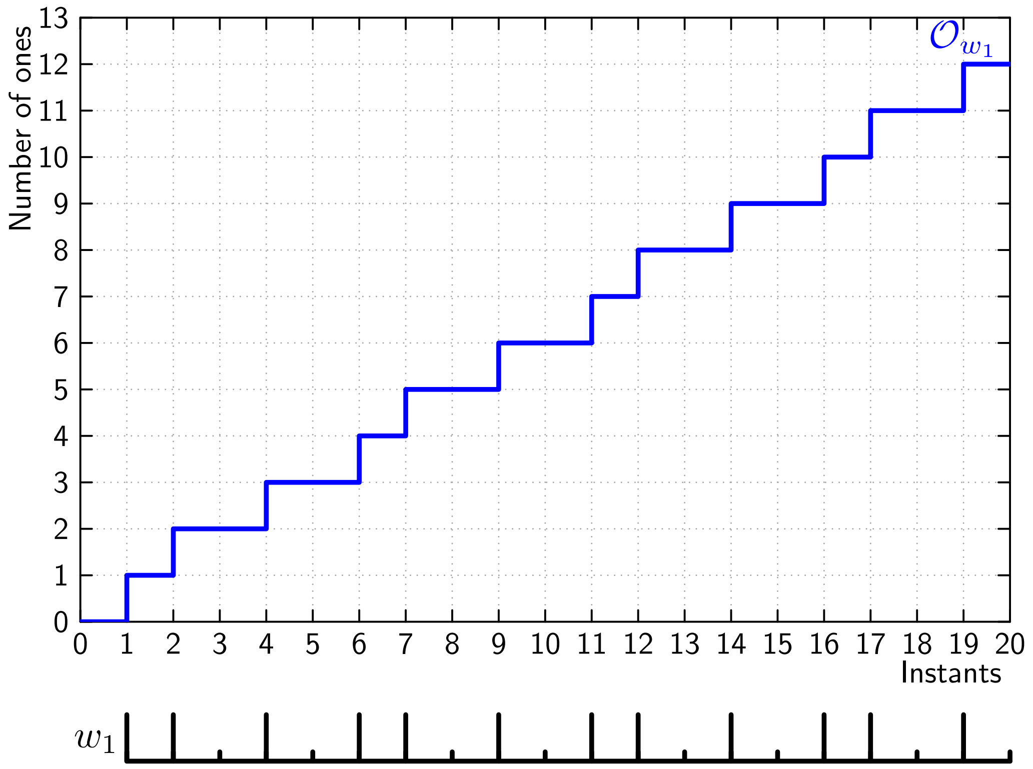
— 1 for presence

— 0 for absence

Definition :

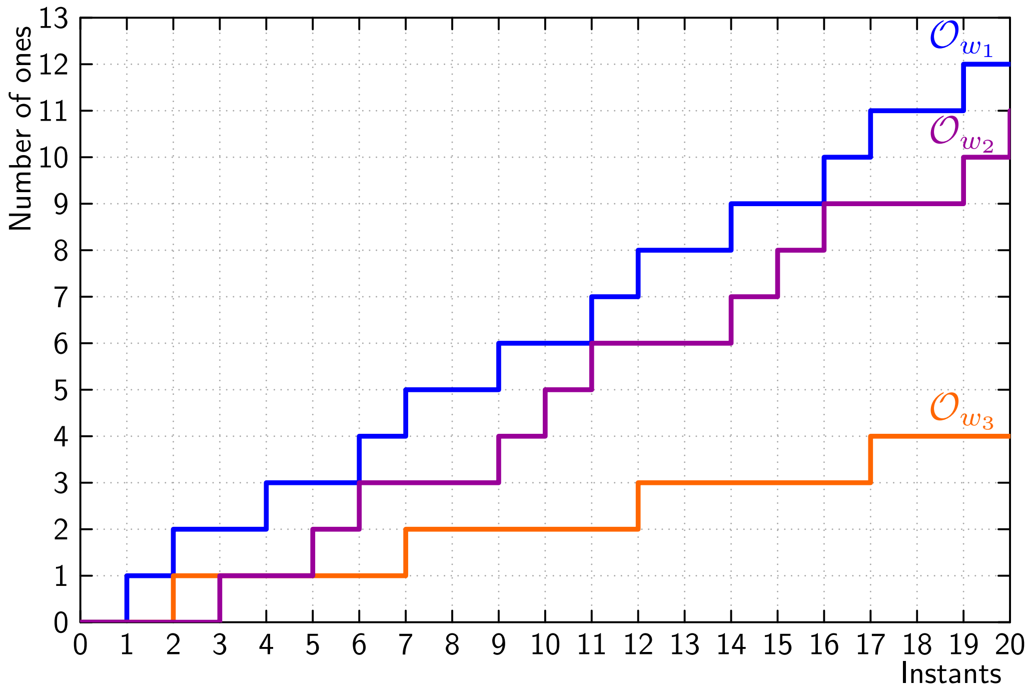
$$w ::= u(v) \quad \text{where } u \in (0+1)^* \text{ and } v \in (0+1)^+$$

Clocks and infinite binary words



$$\mathcal{O}_w(i) = \text{cumulative function of 1 from } w$$

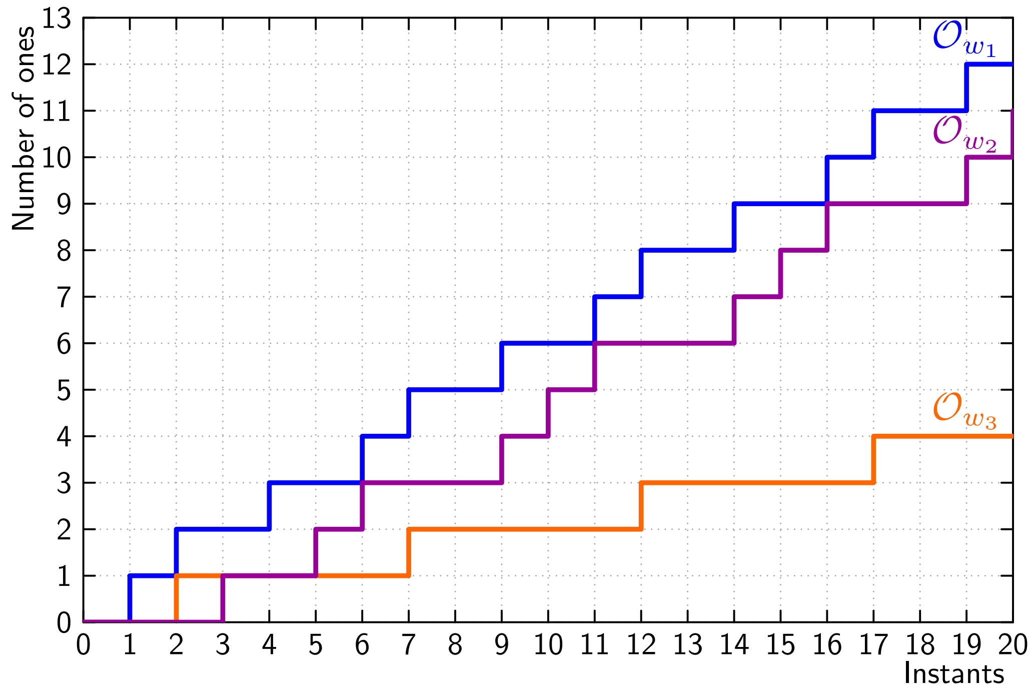
Clocks and infinite binary words



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

sub-typing $w_1 <: w_2 \overset{def}{\Leftrightarrow} \exists n \in \mathbb{N}, \forall i,\ 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n$

Clocks and infinite binary words



buffer	$size(w_1, w_2) = \max_{i \in \mathbb{N}}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$
sub-typing	$w_1 <: w_2 \overset{def}{\Leftrightarrow} \exists n \in \mathbb{N}, \forall i,\ 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n$
synchronizability	$w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists b_1, b_2 \in \mathbb{Z}, \forall i,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$
precedence	$w_1 \preceq w_2 \overset{def}{\Leftrightarrow} \forall i,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$

Multi-clock

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

c on w is a **sub-clock** of c, by moving in w at the pace of c. E.g., $1(10)$ on $(01) = (0100)$.

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)
base on p_1	1	1	0	1	0	1	0	1	0	1	...	1(10)
p_2	0	1		0		1		0		1	...	(01)
(base on p_1) on p_2	0	1	0	0	0	1	0	0	0	1	...	(0100)

For ultimately periodic clocks, precedence, synchronizability and equality are decidable (but expensive)

Come-back to the language

Pure synchrony :

- close to an ML type system (e.g., SCADE 6)
- structural equality of clocks

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

Relaxed Synchrony :

- we add a **sub-typing** rule :

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

- defines synchronization points when a buffer is inserted
- the basis of the language Lucy-N (Plateau and Mandel).

What about non periodic systems ?

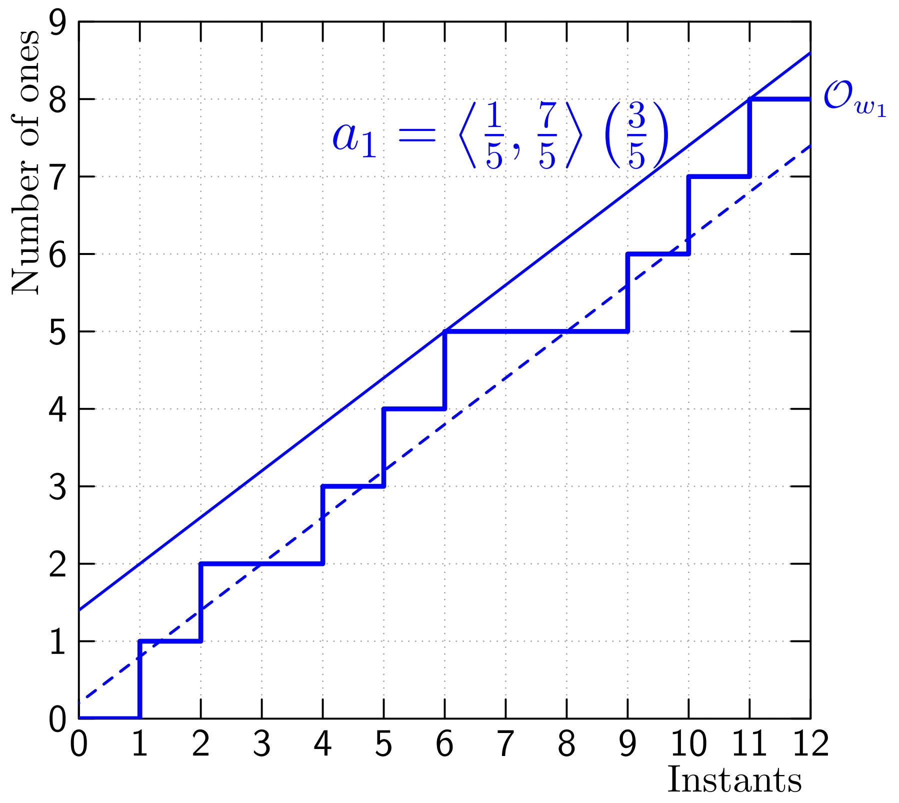
- The same idea : synchrony + properties between clocks. Insuring the absence of deadlocks and bounded buffering.

- The **exact** computation with periodic clocks is expensive. E.g., $(10100100) \texttt{ on } 0^{3600}(1) \texttt{ on } (101001001) =$ $0^{9600}(10^4 10^7 10^7 10^2)$

- Motivations :
  1. To treat long periodic patterns. To avoid an exact computation.
  2. To deal with almost periodic clocks. E.g., $\alpha \texttt{ on } w$ where $w = 00.(\ (10) + (01)\ )^*$ (e.g. $w =$ 00 01 10 01 01 10 01 10 . . .)

Idea : manipulate sets of clocks ; turn questions into arithmetic ones

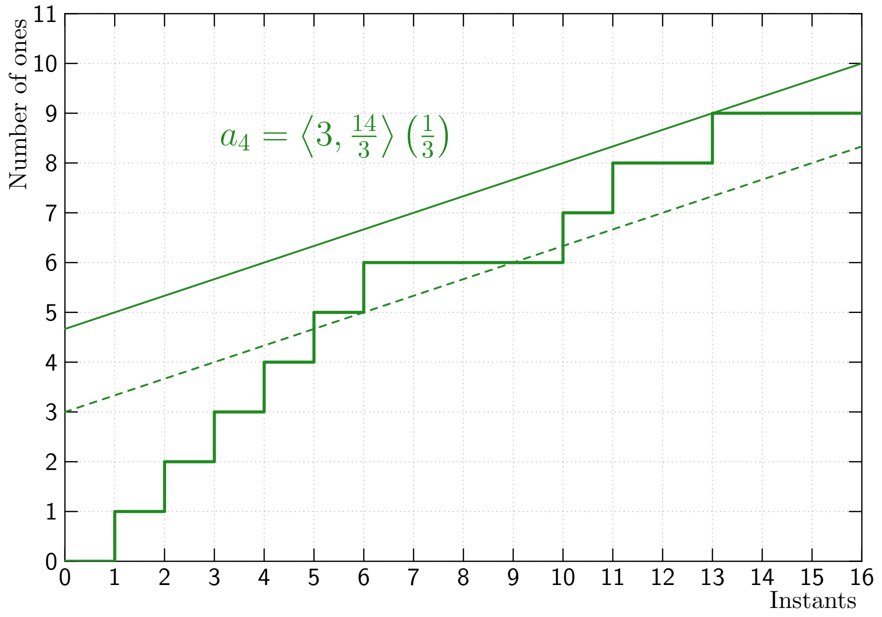
Abstraction of Infinite Binary Words

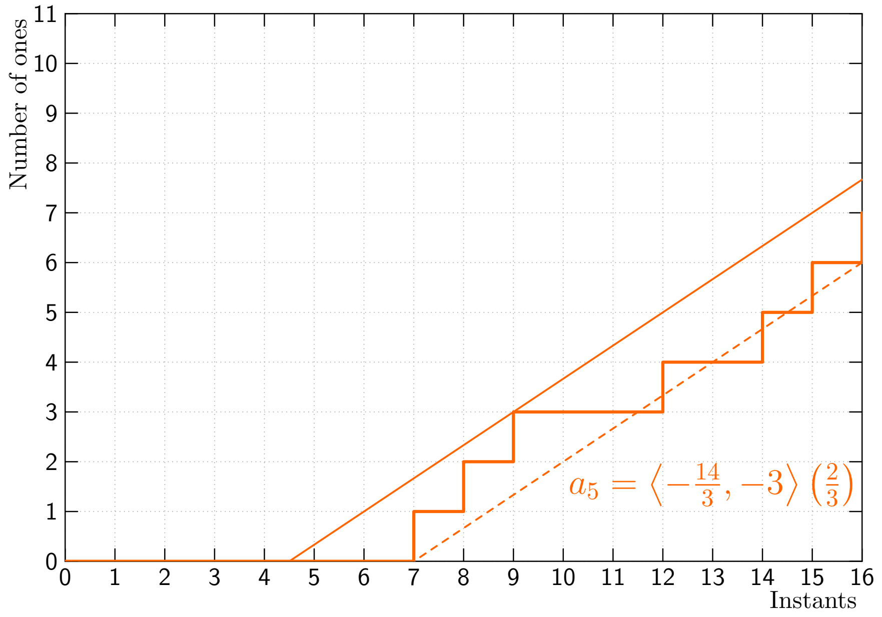


A word w can be abstracted by two lines : $abs(w) = \left\langle b^0, b^1 \right\rangle (r)$

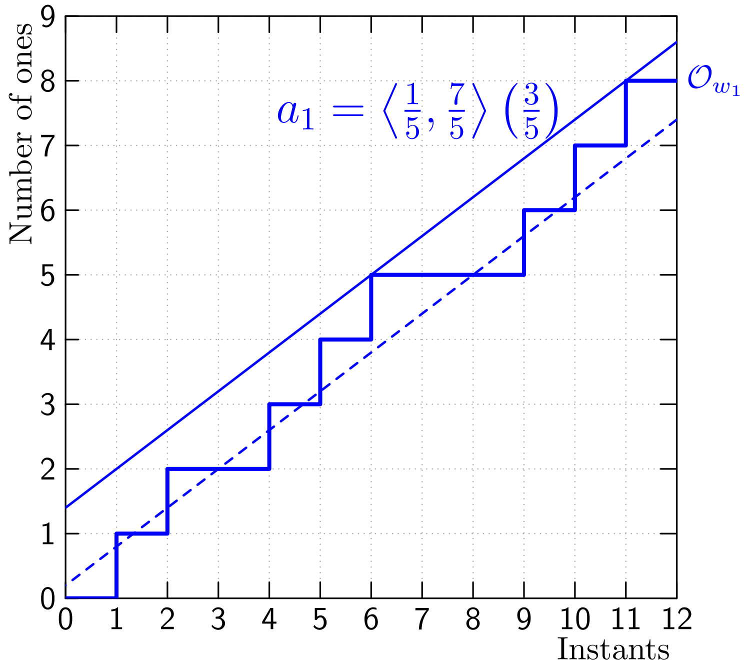
$$concr\left(\left\langle b^0, b^1 \right\rangle (r)\right) \overset{def}{\Leftrightarrow} \left\{ w,\ \forall i \geq 1,\ \wedge \begin{array}{lcl} w[i] = 1 & \Rightarrow & \mathcal{O}_w(i) \leq r \times i + b^1 \\ w[i] = 0 & \Rightarrow & \mathcal{O}_w(i) \geq r \times i + b^0 \end{array} \right\}$$

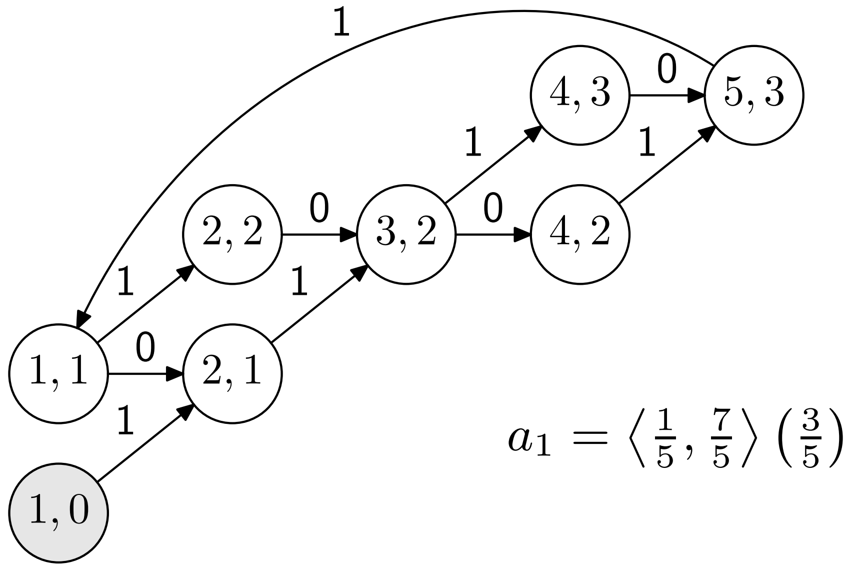
Abstraction of Infinite Binary Words





Abstract Clocks as Automata

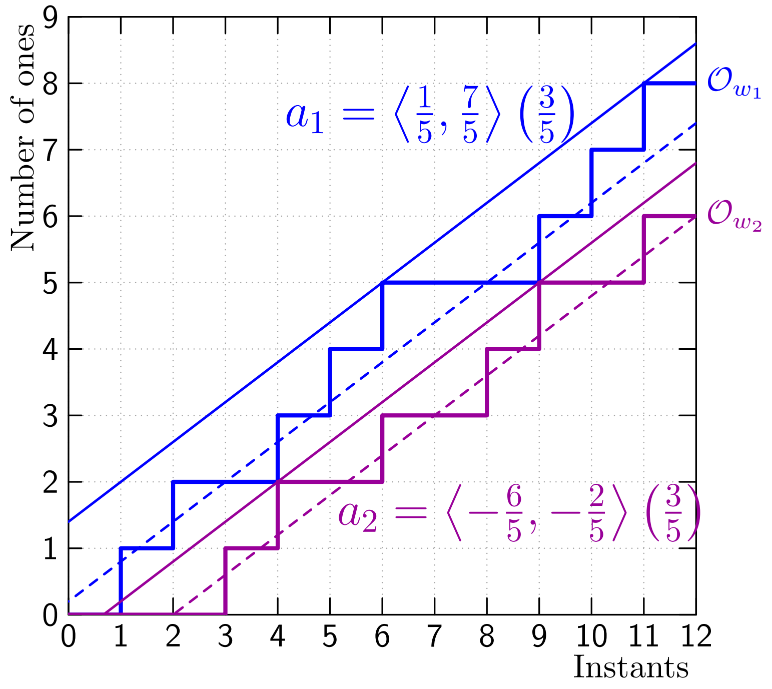




$1,0$
$1,1$
$2,1$
$2,2$
$3,2$
$4,2$
$4,3$
$5,3$
$a_1 = \langle \frac{1}{5}, \frac{7}{5} \rangle \left(\frac{3}{5} \right)$

- set of states $\{(i,j) \in \mathbb{N}^2\}$: coordinates in the 2D-chronogram
- finite number of state equivalence classes
- transition function δ : $\begin{cases} \delta(1,(i,j)) = nf(i+1, j+1) & \text{if } j+1 \leq r \times i + b^1 \\ \delta(0,(i,j)) = nf(i+1, j+0) & \text{if } j+0 \geq r \times i + b^0 \end{cases}$
- allows to check/generate clocks

Abstract Relations



Synchronizability : $r_1 = r_2 \Leftrightarrow \left\langle {b^0}_1, {b^1}_1 \right\rangle (r_1) \bowtie^{\sim} \left\langle {b^0}_2, {b^1}_2 \right\rangle (r_2)$

Precedence : ${b^1}_2 - {b^0}_1 < 1 \Rightarrow \left\langle {b^0}_1, {b^1}_1 \right\rangle (r) \preceq^{\sim} \left\langle {b^0}_2, {b^1}_2 \right\rangle (r)$

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

- ▷ proposition : $abs(w_1) <:^{\sim} abs(w_2) \Rightarrow w_1 <: w_2$
- ▷ buffer : $size(a_1, a_2) = \left\lfloor {b^1}_1 - {b^0}_2 \right\rfloor$

Abstract Operators

Composed clocks : $c ::= w \mid \texttt{not}\ w \mid c\ \texttt{on}\ c$

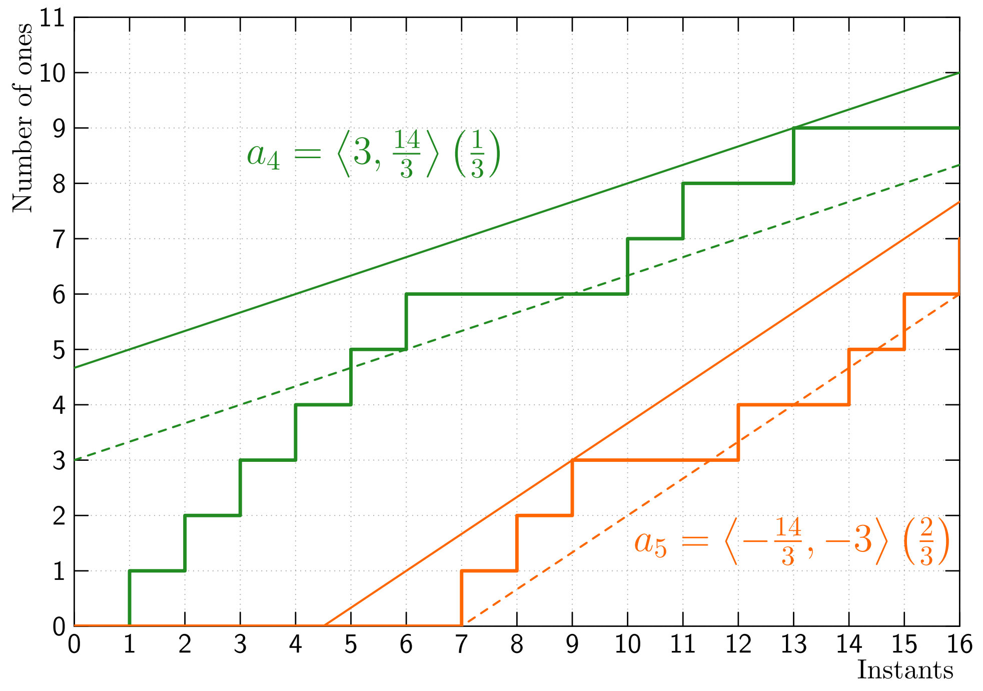
Abstraction of a composed clock :

$$\begin{array}{lcl} abs(\texttt{not}\ w) & = & \texttt{not}^{\sim}\ abs(w) \\ abs(c_1\ \texttt{on}\ c_2) & = & abs(c_1)\ \texttt{on}^{\sim}\ abs(c_2) \end{array}$$

Operators correctness property :

$$\begin{array}{lcl} \texttt{not}\ w & \in & concr(\texttt{not}^{\sim}\ abs(w)) \\ c_1\ \texttt{on}\ c_2 & \in & concr(abs(c_1)\ \texttt{on}^{\sim}\ abs(c_2)) \end{array}$$

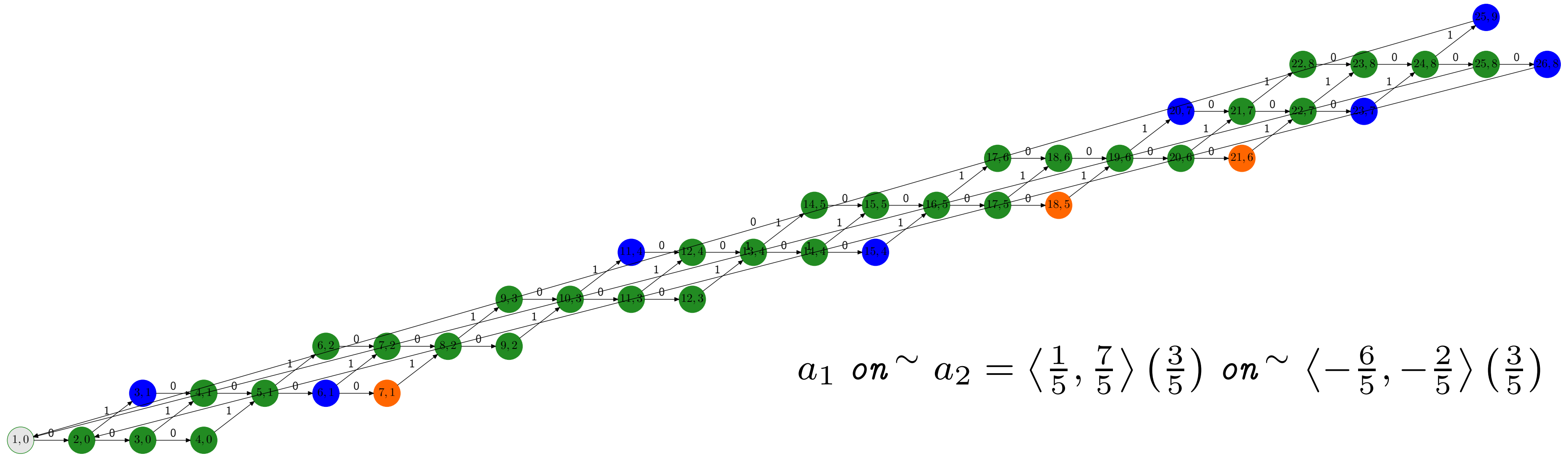
Abstract Operators



$\mathit{not}^{\sim}$ operator definition :

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

Abstract Operators



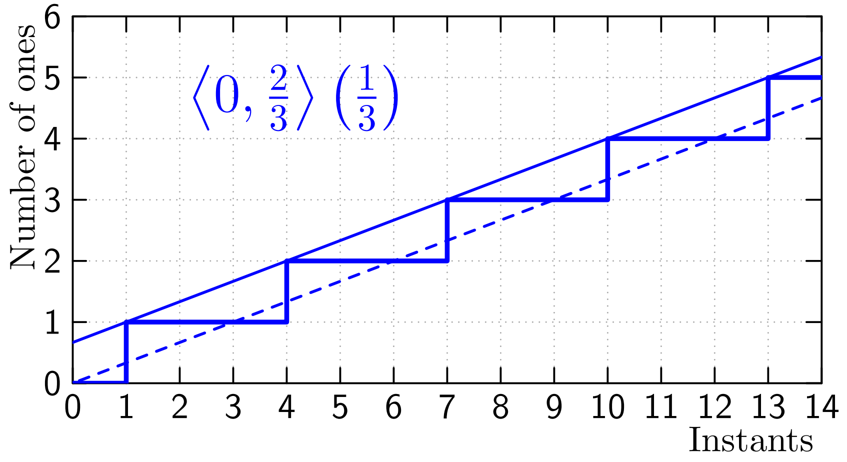
$$a_1 \ \boldsymbol{\mathit{on}}^{\sim} \ a_2 = \left\langle \tfrac{1}{5}, \tfrac{7}{5} \right\rangle \left(\tfrac{3}{5}\right) \ \boldsymbol{\mathit{on}}^{\sim} \ \left\langle -\tfrac{6}{5}, -\tfrac{2}{5} \right\rangle \left(\tfrac{3}{5}\right)$$

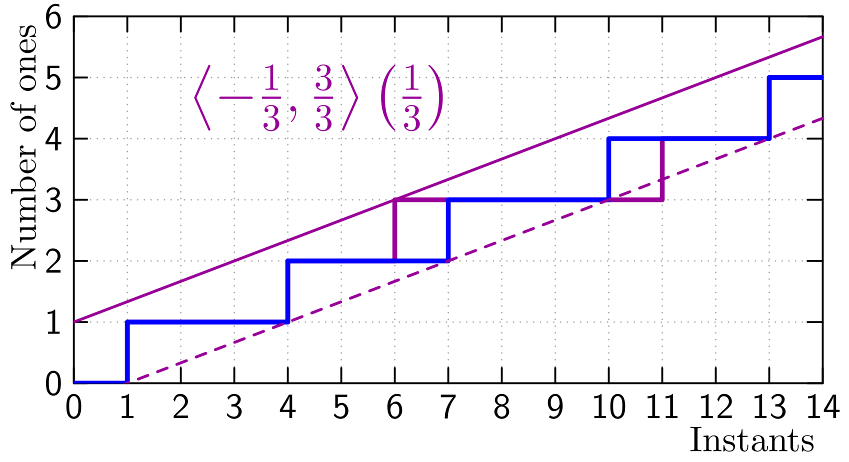
$\boldsymbol{\mathit{on}}^{\sim}$ operator definition :

$$\begin{array}{ccccc}
 & \langle \quad {b^0}_1 & , & {b^1}_1 \quad \rangle (& r_1 \ ) \\
\boldsymbol{\mathit{on}}^{\sim} & \langle \quad {b^0}_2 & , & {b^1}_2 \quad \rangle (& r_2 \ ) \\
= & \langle \, {b^0}_1 \times r_2 + {b^0}_2 & , & {b^1}_1 \times r_2 + {b^1}_2 \, \rangle (& r_1 \times r_2 \,)
\end{array}$$

with $\quad {b^0}_1 \leq 0, \quad {b^0}_2 \leq 0$

Modeling Jitter





- set of clock of rate $r = \frac{1}{3}$ and jitter 1 can be specified by $\langle -\frac{1}{3}, \frac{3}{3} \rangle \left(\frac{1}{3}\right)$
- $\langle -\frac{1}{3}, \frac{3}{3} \rangle \left(\frac{1}{3}\right) = \langle -1, 1 \rangle (1)$ ***on***$^{\sim}$ $\langle 0, \frac{2}{3} \rangle \left(\frac{1}{3}\right)$
- $f :: \forall \alpha. \alpha \rightarrow \alpha$ `on`$^{\sim}$ $\langle -\frac{1}{3}, \frac{3}{3} \rangle \left(\frac{1}{3}\right)$

Formalization in a Proof Assistant

By Louis Mandel and Florence Plateau

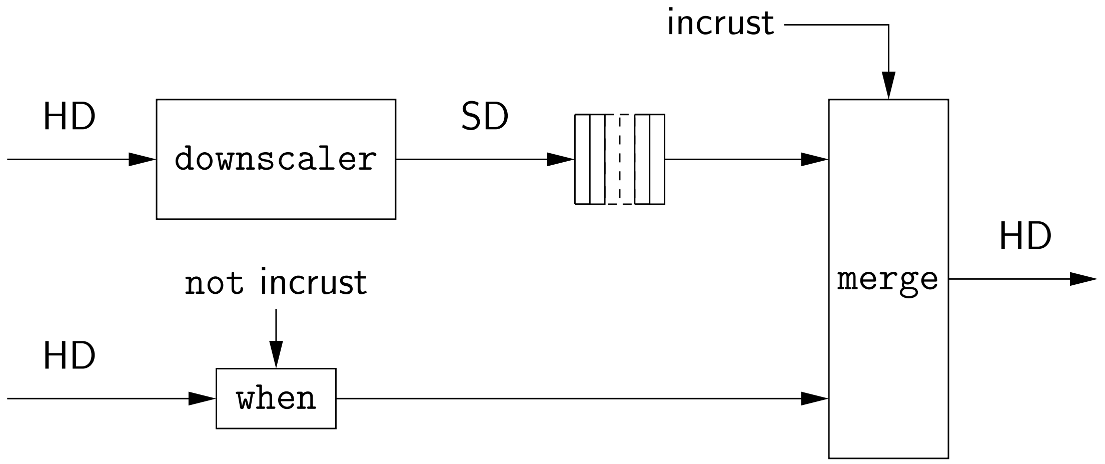
Most of the properties have been proved in Coq

- example of property

```
Property on_absh_correctness:
  forall (w1:ibw) (w2:ibw),
  forall (a1:abstractionh) (a2:abstractionh),
  forall H_wf_a1: well_formed_abstractionh a1,
  forall H_wf_a2: well_formed_abstractionh a2,
  forall H_a1_eq_absh_w1: in_abstractionh w1 a1,
  forall H_a2_eq_absh_w2: in_abstractionh w2 a2,
  in_abstractionh (on w1 w2) (on_absh a1 a2).
```

- number of Source Lines of Code
  - specifications : about 1600 SLOC
  - proofs : about 5000 SLOC

Back to the Picture in Picture Example



- abstraction of downscaler output :

  $abs((10100100)\ \mathit{on}\ 0^{3600}(1)\ \mathit{on}\ (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720}))$

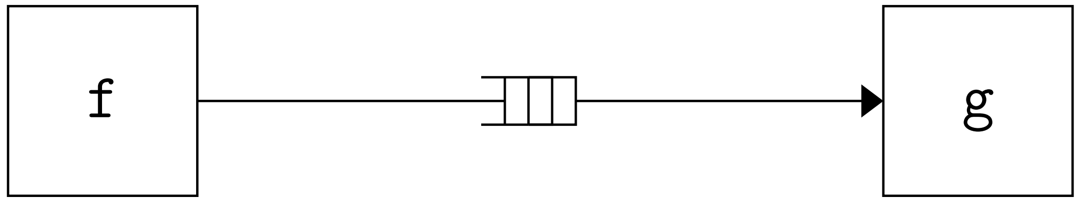
  $= \langle 0, \frac{7}{8} \rangle \left(\frac{3}{8}\right)\ \mathit{on}^{\sim}\ \langle -3600, -3600 \rangle\ (1)\ \mathit{on}^{\sim}\ \langle -400, 480 \rangle \left(\frac{4}{9}\right) = \langle -2000, -\frac{20153}{18} \rangle \left(\frac{1}{6}\right)$
- minimal delay and buffer :

	delay	buffer size
exact result	9 598 (≈ time to receive 5 HD lines)	192 240 (≈ 267 SD lines)
abstract result	11 995 (≈ time to receive 6 HD lines)	193 079 (≈ 268 SD lines)

This is implemented in Lucy-N `http://lucy-n.org` by Louis Mandel.

Parallel implementation and integer clocks

Parallel processes communicating through a buffer



```
int f_out;
while (1) {
  f_step (f_mem, &f_out);
  fifo.push(f_out);
}
```

```
int g_in;
while (1) {
  fifo.pop(&g_in);
  v = g_step (g_mem, g_in);
}
```

Buffers allow to desynchronize the execution

FIFO with batching

To pop, the consumer has to check for the availability of data. This check is expensive. It is better to communicate by chunks.

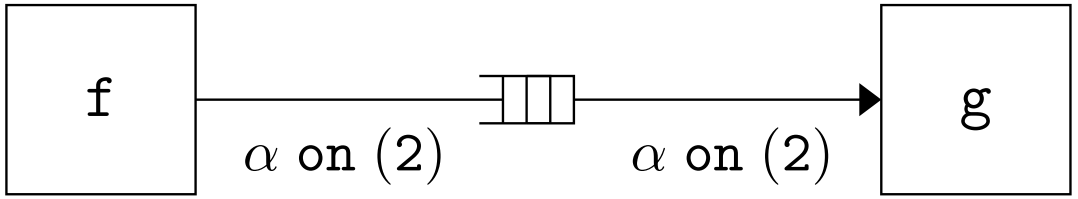
Batch :

- the consumer can read in the fifo only when *batch* values are available
- the producer can write in the fifo only when *batch* rooms are available

Batch size :	001	Cycles/push :	23.07	Bandwidth :	589.45 MB/s
Batch size :	002	Cycles/push :	15.79	Bandwidth :	861.40 MB/s
Batch size :	004	Cycles/push :	12.06	Bandwidth :	1127.83 MB/s
Batch size :	008	Cycles/push :	10.00	Bandwidth :	1359.69 MB/s
Batch size :	016	Cycles/push :	7.51	Bandwidth :	1810.58 MB/s
Batch size :	032	Cycles/push :	7.33	Bandwidth :	1855.32 MB/s
Batch size :	064	Cycles/push :	7.33	Bandwidth :	1855.20 MB/s

Batching : reduce the synchronization with the FIFO

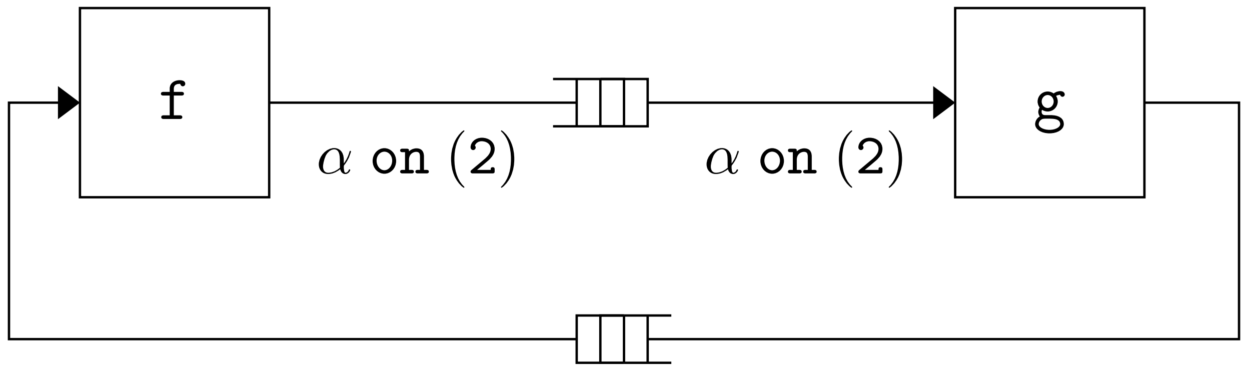
Integer clocks



Burst :

- allows to compute and communicate several values within one instant
- formulas can be easily lifted to integers

Integer clocks



Burst :

- allows to compute several values into one instant
- formulas can be easily lifted to integers
- impacts causality

This has been studied by Adrien Guatto in his PhD. thesis (2016).

Type based clock calculus

Lucid Synchrone

— stream Kahn semantics, clocks, functions possibly higher-order
— study (implement) extensions of Lustre
— experiment things, manage all the compilation chain and write programs !
— Version 1 (1995), Version 2 (2001), V3 (2006)

Quite fruitful :

— the Scade 6 language and its compiler (first release in 2008) incorporates several features from Lucid Synchrone
— the LCM language at Dassault-Systm̀es (Delmia Automation) based on the same principles
— several features reused in Stimulus, a language for requirement simulation.

Références

[1] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.di.ens.fr/∼pouzet/bib/bib.html`.

[2] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94 :125–140, 1992.

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

[4] 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.

[5] 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.

[6] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[7] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n : a n-Synchronous Extension of Lustre. In *10th International Conference on Mathematics of Program Construction (MPC'10)*, Manoir St-Castin, Québec, Canada, June 2010. Springer LNCS.

[8] Louis Mandel, Florence Plateau, and Marc Pouzet. Static Scheduling of Latency Insensitive Designs with Lucy-n. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Austin, Texas, USA, October 30 – November 2 2011.

[9] Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Université Paris-Sud 11, Orsay, France, 6 janvier 2010.