Clocks in Kahn Process Networks

Marc Pouzet
École normale supérieure

Course notes
April 2020
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:

\[
\begin{align*}
  z, t &= Q(y) \\
  y &= P(x, r) \\
  r &= M(t) \\
  z &= F(x)
\end{align*}
\]

What is the meaning (semantics) of these two sets of equations? 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:

```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`.
```
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 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)
Pros and Cons of KPN

(+) : **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.

\[
\begin{array}{ccccccc}
  x & = & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
  f(x) & = & y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  f(x) & = & y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
\end{array}
\]

This appeared to be a useful model for video apps (TV boxes) : Sally (Philips NatLabs), StreamIt (MIT), Xstream (ST-micro) with various “synchronous” restriction à la **SDF** (Edward Lee)
A small dataflow kernel

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

\[
e ::= e \text{ fby } e \mid op(e, ..., e) \mid x \mid v \\
\hspace{100pt} \mid \text{merge } e e e \mid e \text{ when } e
\]

\[
op ::= + \mid - \mid \text{not} \mid ...
\]

Definition of stream functions, equations:

\[
d ::= \text{node } f(p) = p \text{ with } D
\]

\[
p ::= x, ..., x \mid x
\]

\[
D ::= x = e \mid D \text{ and } D \mid \text{var } x \text{ in } D
\]
Dataflow Primitives

<table>
<thead>
<tr>
<th></th>
<th>( x )</th>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>( y_0 )</td>
<td>( y_1 )</td>
<td>( y_2 )</td>
<td>( y_3 )</td>
<td>( y_4 )</td>
<td>( y_5 )</td>
<td></td>
</tr>
<tr>
<td>( x + y )</td>
<td>( x_0 + y_0 )</td>
<td>( x_1 + y_1 )</td>
<td>( x_2 + y_2 )</td>
<td>( x_3 + y_3 )</td>
<td>( x_4 + y_4 )</td>
<td>( x_5 + y_5 )</td>
<td></td>
</tr>
<tr>
<td>( x \ fby \ y )</td>
<td>( x_0 )</td>
<td>( y_0 )</td>
<td>( y_1 )</td>
<td>( y_2 )</td>
<td>( y_3 )</td>
<td>( y_4 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( h )</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x' = x \text{ when } h )</td>
<td>( x_0 )</td>
<td>( x_2 )</td>
<td>( x_4 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( z )</th>
<th>( z_0 )</th>
<th>( z_1 )</th>
<th>( z_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge ( h \ x' z )</td>
<td>( x_0 )</td>
<td>( z_0 )</td>
<td>( x_2 )</td>
</tr>
</tbody>
</table>

Sampling:

- If \( h \) is a boolean sequence, \( x \text{ when } h \) 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 $V$. $V^* = \cup_{n=0}^{\infty} V^n$ is the Kleene star operation.

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

— $\epsilon$ 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, $\epsilon$ 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 $\text{lift}^2(+) (s_1, s_2)$

\(^a\) The minimal element is usually written $\perp$. 

Course notes - Marc Pouzet

12/64
\[ \text{lift}^0(v) = v.\text{lift}^0(v) \]
\[ \text{lift}^1(\text{op})(v.s) = \text{op}(v).\text{lift}^1(\text{op})(s) \]
\[ \text{lift}^1(\text{op})(\epsilon) = \epsilon \]
\[ \text{lift}^2(\text{op})(v_1.s_1, v_2.s_2) = \text{op}(v_1, v_2).\text{lift}^2(\text{op})(s_1, s_2) \]
\[ \text{lift}^2(\text{op})(s_1, s_2) = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \]
\[ \text{fby}(s_1)(s_2) = \epsilon \text{ if } s_1 = \epsilon \]
\[ \text{fby}(v_1.s_1)(s_2) = v_1.s_2 \]
Kahn Semantics

\[
\begin{align*}
\text{when}(v.s, 1.c) &= v.\text{when}(s, c) \\
\text{when}(v.s, 0.c) &= \text{when}(s, c) \\
\text{when}(s_1, s_2) &= \varepsilon \text{ if } s_1 = \varepsilon \text{ or } s_2 = \varepsilon \\
\text{merge}(1.c, v.s_1, s_2) &= v.\text{merge}(c, s_1, s_2) \\
\text{merge}(0.c, s_1, v.s_2) &= v.\text{merge}(c, s_1, s_2) \\
\text{merge}(1.c, \varepsilon, s_2) &= \varepsilon \\
\text{merge}(0.c, s_1, \varepsilon) &= \varepsilon \\
\text{merge}(\varepsilon, s_1, s_2) &= \varepsilon
\end{align*}
\]

All those functions are continuous [2].
An other formulation

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:

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

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 $\phi_A$ between an initial segment $I_A$ of $\mathbb{N}$ and $A$.

A signal $u$ is a sequence $(u_n)_{n \in \mathbb{N}}$, finite or not, indexed on an initial segment $N$. 
\[
\begin{align*}
lift^0(v) &= (u)_{n \in \mathbb{N}} \quad \text{with} \quad \forall n \in \mathbb{N}. u_n = v \\
lift^1(op)((u_n)_{n \in \mathbb{N}}) &= (v_n)_{n \in \mathbb{N}} \quad \text{with} \quad \forall n \in \mathbb{N}. v_n = op(v_n) \\
lift^2(op)((u_n)_{n \in \mathbb{N}}, (v_n)_{n \in \mathbb{N}}) &= (w_n)_{n \in \mathbb{N}} \quad \text{with} \quad \forall n \in \mathbb{N}. w_n = op(u_n, v_n) \\
fby((u_n)_{n \in \mathbb{N}})((v_n)_{n \in \mathbb{N}}) &= (w_n)_{n \in \mathbb{N}} \quad \text{with} \quad w_0 = u_0 \\
& \quad \text{and} \quad \forall n \in \mathbb{N}\setminus\{0\}. w_n = v_{n-1}
\end{align*}
\]

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

\[
N_h = \{ k \in \mathbb{N} \mid h_k = 1 \}
\]

and

\[
N_{\overline{h}} = \{ k \in \mathbb{N} \mid h_k = 0 \}
\]

\(N_h\) with \(N_{\overline{h}}\) form a partition of \(\mathbb{N}\).
when\((\seq{u}, \seq{h})\) = \(\seq{v}\) with \(v_n = u_{\phi_{Nh}}(n)\)
merge\((\seq{h}, \seq{u}_{Nh}, \seq{v}_{Nh})\) = \(\seq{w}\) with \(w_n = u_n\) if \(n \in Nh\)

and \(w_n = v_n\) if \(n \in \overline{Nh}\)

The base clock is the constant sequence \(base\) such that \(\forall n \in \mathbb{N}, base_n = 1\).
Making it a bit more operational

constant \( v \ n = v \)

\[ \text{lift1} \ op \ x \ n = \text{op}(x(n)) \]
\[ \text{notl} \ x = \text{lift1} \ \text{not} \]

\[ \text{lift2} \ op \ x \ y \ n = \text{op}(x(n)) (y(n)) \]
\[ \text{lift3} \ op \ x \ y \ z \ n = \text{op}(x(n)) (y(n)) (z(n)) \]

\[ \text{x fby y} \ 0 = x(0) \]
\[ \text{x fby y} \ n = y(n-1) \]

\[ \text{x when} \ h \ n = x(I(h)(n+1)) \]
\[ \text{merge} \ h \ x \ y \ n = \text{if} \ h(n) \ \text{then} \ x(O(h)(n)) \ \text{else} \ y(O(\text{notl} \ h)(n)) \]

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 \( h \).

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

\[
O(h)(n) = h(n) + (\text{if } n = 0 \text{ then } 0 \text{ else } O(h)(n-1))
\]

\[
I(h)(n) = I'(h)(0)(n)
\]

\[
I'(h)(i)(n) = \begin{cases} 
  h(i) & \text{if } n = 1 \\
  \text{if } I'(h)(i+1)(n-1) & \text{else if } h(i) \text{ then } n = 1 \\
  \text{else } I'(h)(i+1)(n) & \text{else}
\end{cases}
\]

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

-- lifting constants
constant x = x : (constant x)

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

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

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

merge (True : c) (x : xs) y = x : (merge c xs y)
merge (False : c) x (y : ys) = 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 useful 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 greater 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 $(o_n)_{n \in \mathbb{N}}$ 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)

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

$$u_1 = x$$

$\eps = \text{constant } 0.001$

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:**

\[ \bot \leq_{\text{struct}} v, (v : w) \leq_{\text{struct}} (v' : w') \text{ iff } v \leq_{\text{struct}} v' \text{ and } w \leq_{\text{struct}} w'. \]

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

\[
\begin{align*}
\text{hd} & (x:y) = x \\
\text{tl} & (x:y) = y \\
\text{incr} & (x:y) = (x+1) : \text{incr} \ y \\
\text{one} & = 1 : \text{one} \\
x & = \left(\text{if } \text{hd}(\text{tl}(\text{tl}(\text{tl}(x)))) = 5 \text{ then } 3 \text{ else } 4\right) : 1 : 2 : 3 : \text{one} \\
\text{output} & = \left(\text{hd}(\text{tl}(\text{tl}(\text{tl}(x))))\right) : \left(\text{hd}(\text{tl}(\text{tl}(x)))\right) : \left(\text{hd}(x)\right) : \text{output}
\end{align*}
\]

The values are:

\[
\begin{align*}
- x & = 4 : 1 : 2 : 3 : 1 : \ldots \\
- \text{output} & = 3 : 2 : 4 : 3 : 2 : 4 : \ldots
\end{align*}
\]
These streams may be constructed lazily:
- \( x^0 = \bot, x^1 = \bot : 1 : 2 : 3 : un, x^2 = 4 : 1 : 2 : 3 : one. \)
- \( output^0 = \bot, output^1 = 3 : 2 : 4 : \ldots \)

An other example (due to Paul Caspi):

\[
\text{nat} = \text{zero} \ 'fby' \ (\text{incr} \ \text{nat}) \\
\text{ifn} \ n \ x \ y = \text{if} \ n \ \leq \ 9 \ \text{then} \ \text{hd}(x) : \ \text{ifn} \ (n+1) \ (\text{tl}(x)) \ (\text{tl}(y)) \ \text{else} \ y \\
\text{if9} \ x \ y = \text{ifn} \ 0 \ x \ y
\]

\[
x = \text{if9} \ (\text{incr} \ (\text{tl} \ x)) \ \text{nat}
\]

We have \( x = 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 11, 12, 13, 14, 15, \ldots. \)

Are they reasonable 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:

\[ half = 0.\perp.0.\perp... \]

\[
\begin{align*}
\text{fail} &= \text{fail} \\
\text{half} &= 0:\text{fail}:\text{half} \\
\text{fill } x &= (\text{hd}(x)) : \text{fill } (\text{tl}(\text{tl } x)) \\
\text{ok} &= \text{fill } \text{half}
\end{align*}
\]

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:**

\[
x \leq y \text{ if } x \text{ is a prefix of } y, \text{ that is: } \perp \leq x \text{ and } v.x \leq v.y \text{ if } x \leq y
\]

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

```haskell
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 \( \bot \) (stack overflow).
Some “synchrony” monsters

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

**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

$$IB = \{0, 1\}$$

$$CLOCK = IB_{\infty}$$

$$\text{clock } \epsilon = \epsilon$$

$$\text{clock } (abs.x) = 0.\text{clock } x$$

$$\text{clock } (v.x) = 1.\text{clock } x$$

**Synchronous streams**:

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

**An other possible encoding**: $x : (V \times \mathbb{N})_{\infty}$
Dataflow Primitives

Constant :

\[ i\#(\epsilon) = \epsilon \]
\[ i\#(1.cl) = i.i\#(cl) \]
\[ i\#(0.cl) = abs.i\#(cl) \]

Point-wise application :

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

\[ +\#(s_1, s_2) = \epsilon \text{ if } s_i = \epsilon \]
\[ +\#(abs.s_1, abs.s_2) = abs.+\#(s_1, s_2) \]
\[ +\#(v_1.s_1, v_2.s_2) = (v_1 + v_2).+\#(s_1, s_2) \]
Partial definitions

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

Constraint their domain:

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

- (+): int \times int \rightarrow 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

\[
\begin{align*}
  s_1 \text{ when} # s_2 & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
  (\text{abs}.s) \text{ when} # (\text{abs}.c) & = \text{abs}.s \text{ when} # c \\
  (v.s) \text{ when} # (1.c) & = v.s \text{ when} # c \\
  (v.s) \text{ when} # (0.c) & = \text{abs}.x \text{ when} # c \\
  \text{merge } c s_1 s_2 & = \epsilon \text{ if one of the } s_i = \epsilon \\
  \text{merge } (\text{abs}.c)(\text{abs}.s_1)(\text{abs}.s_2) & = \text{abs}.\text{merge } c s_1 s_2 \\
  \text{merge } (1.c)(v.s_1)(\text{abs}.s_2) & = v.\text{merge } c s_1 s_2 \\
  \text{merge } (0.c)(\text{abs}.s_1)(v.s_2) & = v.\text{merge } c s_1 s_2
\end{align*}
\]
Examples

<table>
<thead>
<tr>
<th>$base = (1)$</th>
<th>1 1 1 1 1 1 1 1 1 1 1 1 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x_0$  $x_1$  $x_2$  $x_3$  $x_4$  $x_5$  $x_6$  $x_7$  $x_8$  $x_9$  $x_{10}$  $x_{11}$ ...</td>
</tr>
<tr>
<td>$h = (10)$</td>
<td>1 0 1 0 1 0 1 0 1 0 1 0 ...</td>
</tr>
<tr>
<td>$y = x$ when $h$</td>
<td>$x_0$  $x_2$  $x_4$  $x_6$  $x_8$  $x_{10}$  $x_{11}$ ...</td>
</tr>
<tr>
<td>$h' = (100)$</td>
<td>1 0 0 1 0 0 1 0 0 1 0 0 ...</td>
</tr>
<tr>
<td>$z = y$ when $h'$</td>
<td>$x_0$  $x_6$  $x_{11}$ ...</td>
</tr>
<tr>
<td>$k$</td>
<td>$k_0$  $k_1$  $k_2$  $k_3$  ...</td>
</tr>
<tr>
<td>merge $h'$ z k</td>
<td>$x_0$  $k_0$  $k_1$  $x_6$  $k_2$  $k_3$  ...</td>
</tr>
</tbody>
</table>

let clock five =
    let rec f = true fby false 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

- $x \text{ when}^\# y$ is defined when $x$ and $y$ have the same clock $cl$
- the clock of $x \text{ when}^\# c$ is written $cl$ on $c$ : “$c$ moves at the pace of $cl$”

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

We get:

$$
\begin{align*}
\text{when} : \forall cl. \forall x : cl. \forall c : cl.cl \text{ on } c \\
\text{merge} : \forall cl. \forall c : cl. \forall x : cl \text{ on } c. \forall y : cl \text{ on } \text{not } c.cl
\end{align*}
$$

Written instead:

$$
\begin{align*}
\text{when} : \forall cl.cl \rightarrow (c : cl) \rightarrow cl \text{ on } c \\
\text{merge} : \forall cl.(c : cl) \rightarrow cl \text{ on } c \rightarrow cl \text{ on } \text{not } c \rightarrow cl
\end{align*}
$$
Checking Synchrony

The previous program is now rejected.

This is a now a **typing error**

```haskell
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**
— 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

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

- **downscaler**: reduction of an HD image \((1920 \times 1080 \text{ pixels})\) to an SD image \((720 \times 480 \text{ pixels})\)
- **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
— based on the use of *infinite ultimately periodic sequences*
— a precedence relation $cl_1 <: cl_2$
Ultimately periodic sequences

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

\[
\begin{align*}
(01) &= 01 \ 01 \ 01 \ 01 \ 01 \ 01 \ 01 \ 01 \ldots \\
0(1101) &= 0 \ 1101 \ 1101 \ 1101 \ 1101 \ 1101 \ 1101 \ 1101 \ldots
\end{align*}
\]

— 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

$$\mathcal{O}_{w_1}(i) = \text{cumulative function of 1 from } w$$
buffer \[ \text{size}(w_1, w_2) = \max_{i \in \mathbb{N}} (\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)) \]

sub-typing
\[ w_1 <: w_2 \iff \exists n \in \mathbb{N}, \forall i, 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n \]
Clocks and infinite binary words

buffer

\[\text{size}(w_1, w_2) = \max_{i \in \mathbb{N}} (O_{w_1}(i) - O_{w_2}(i))\]

sub-typing

\[w_1 <: w_2 \iff \exists n \in \mathbb{N}, \forall i, 0 \leq O_{w_1}(i) - O_{w_2}(i) \leq n\]

synchronizability

\[w_1 \bowtie w_2 \iff \exists b_1, b_2 \in \mathbb{Z}, \forall i, b_1 \leq O_{w_1}(i) - O_{w_2}(i) \leq b_2\]

precedence

\[w_1 \preceq w_2 \iff \forall i, O_{w_1}(i) \geq O_{w_2}(i)\]
Multi-clock

\[ c ::= \ w \mid c \text{ on } w \quad w \in (0 + 1)\omega \]

\( c \text{ on } w \) is a **sub-clock** of \( c \), by moving in \( w \) at the pace of \( c \). E.g.,
\[ 1(10) \text{ on } (01) = (0100). \]

<table>
<thead>
<tr>
<th>base</th>
<th>1 1 1 1 1 1 1 1 1 1 ...</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>1 1 0 1 0 1 0 1 0 1 ...</td>
<td>1(10)</td>
</tr>
<tr>
<td>base on ( p_1 )</td>
<td>1 1 0 1 0 1 0 1 0 1 ...</td>
<td>1(10)</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>0 1 0 1 0 1 0 1 ...</td>
<td>(01)</td>
</tr>
<tr>
<td>(base on ( p_1 )) on ( p_2 )</td>
<td>0 1 0 0 0 1 0 0 0 1 ...</td>
<td>(0100)</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
H \vdash e_1 : ck & \quad H \vdash e_2 : ck \\
\hline
\end{align*}
\]

\[
H \vdash op(e_1, e_2) : ck
\]

Relaxed Synchrony:

- we add a sub-typing rule:

\[
\begin{align*}
H \vdash e : ck & \qquad w <: w' \\
\text{(SUB)} & \\
\hline
H \vdash buffer(e) : ck \; on \; w'
\end{align*}
\]

- 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.,

\[(10100100) \text{ on } 0^{3600}(1) \text{ on } (101001001) = 0^{9600}(1^4 10^7 10^7 10^2)\]

- Motivations:
  1. To treat long periodic patterns. To avoid an exact computation.
  2. To deal with almost periodic clocks. E.g., \(\alpha\) on \(w\) where

\[
w = 00.( (10) + (01) )^*\]

(e.g. \(w = 00 01 10 01 01 10 01 10 \ldots\) )

**Idea**: manipulate sets of clocks; turn questions into arithmetic ones
Abstraction of Infinite Binary Words

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

$\text{concr} \left( \langle b^0, b^1 \rangle (r) \right) \overset{\text{def}}{\iff} \left\{ \begin{array}{l} w, \ \forall i \geq 1, \ \wedge \ w[i] = 1 \ \Rightarrow \ O_w(i) \leq r \times i + b^1 \\ w[i] = 0 \ \Rightarrow \ O_w(i) \geq r \times i + b^0 \end{array} \right\}$
Abstraction of Infinite Binary Words

\[ a_4 = \langle 3, \frac{14}{3} \rangle \left( \frac{1}{3} \right) \]

\[ a_5 = \langle -\frac{14}{3}, -3 \rangle \left( \frac{2}{3} \right) \]
Abstract Clocks as Automata

\[ a_1 = \left\langle \frac{1}{5}, \frac{7}{5} \right\rangle \left( \frac{3}{5} \right) \]

\[ \omega_1 \]

\[ O \]

set of states \( \{ (i, j) \in \mathbb{N}^2 \} \): coordinates in the 2D-chronogram

finite number of state equivalence classes

transition function \( \delta \):

\[
\delta(1, (i, j)) = \text{nf}(i + 1, j + 1) \quad \text{if } j + 1 \leq r \times i + b^1 \\
\delta(0, (i, j)) = \text{nf}(i + 1, j + 0) \quad \text{if } j + 0 \geq r \times i + b^0
\]

allows to check/generate clocks
Abstract Relations

Synchronizability : \( r_1 = r_2 \iff \langle b^0_1, b^1_1 \rangle (r_1) \bowtie \sim \langle b^0_2, b^1_2 \rangle (r_2) \)

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

Subtyping : \( a_1 <:\sim a_2 \iff a_1 \bowtie \sim a_2 \land a_1 \preceq \sim a_2 \)

▷ proposition : \( \text{abs}(w_1) <:\sim \text{abs}(w_2) \Rightarrow w_1 <: w_2 \)

▷ buffer : \( \text{size}(a_1, a_2) = \left\lfloor b^1_1 - b^0_2 \right\rfloor \)
Abstract Operators

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

Abstraction of a composed clock:

\[
\begin{align*}
\mathit{abs}(\mathit{not}\ w) & = \mathit{not}\sim \mathit{abs}(w) \\
\mathit{abs}(c_1 \mathit{on}\ c_2) & = \mathit{abs}(c_1) \mathit{on}\sim \mathit{abs}(c_2)
\end{align*}
\]

Operators correctness property:

\[
\begin{align*}
\mathit{not}\ w & \in \mathit{concr}(\mathit{not}\sim \mathit{abs}(w)) \\
c_1 \mathit{on}\ c_2 & \in \mathit{concr}(\mathit{abs}(c_1) \mathit{on}\sim \mathit{abs}(c_2))
\end{align*}
\]
Abstract Operators

\[ a_4 = \left\langle 3, \frac{14}{3} \right\rangle \left( \frac{1}{3} \right) \]

\[ a_5 = \left\langle -\frac{14}{3}, -3 \right\rangle \left( \frac{2}{3} \right) \]

\textit{not} \sim \quad \text{operator definition :}

\[ \textit{not} \sim \left\langle b^0, b^1 \right\rangle (r) = \left\langle -b^1, -b^0 \right\rangle \left( 1 - r \right) \]
\( a_1 \ on^\sim \ a_2 = \langle \frac{1}{5}, \frac{7}{5} \rangle (\frac{3}{5}) \ on^\sim \ \langle -\frac{6}{5}, -\frac{2}{5} \rangle (\frac{3}{5}) \)

\( on^\sim \) operator definition :

\[
\langle \ b^0_1 \ , \ b^1_1 \ \rangle \ (\ r_1 \ )
\]

\[
\langle \ b^0_2 \ , \ b^1_2 \ \rangle \ (\ r_2 \ )
\]

\[
= \langle \ b^0_1 \times r_2 + b^0_2 \ , \ b^1_1 \times r_2 + b^1_2 \ \rangle \ (\ r_1 \times r_2 \ )
\]

with \( b^0_1 \leq 0, \ b^0_2 \leq 0 \)
Modeling Jitter

- set of clock of rate \( r = \frac{1}{3} \) and jitter 1 can be specified by \( \langle -\frac{1}{3}, \frac{3}{3} \rangle \left( \frac{1}{3} \right) \)
- \( \langle -\frac{1}{3}, \frac{3}{3} \rangle \left( \frac{1}{3} \right) = \langle -1, 1 \rangle \left( \frac{1}{3} \right) \) 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:
  \[
  \forall (w_1 : \text{ibw}) (w_2 : \text{ibw}),
  \forall (a_1 : \text{abstractionh}) (a_2 : \text{abstractionh}),
  \forall H_{\text{wf}_a_1} : \text{well_formed_abstractionh} a_1,
  \forall H_{\text{wf}_a_2} : \text{well_formed_abstractionh} a_2,
  \forall H_{a_1\text{eq_absh}_w_1} : \text{in_abstractionh} w_1 a_1,
  \forall H_{a_2\text{eq_absh}_w_2} : \text{in_abstractionh} w_2 a_2,
  \text{in_abstractionh} (\text{on} w_1 w_2) (\text{on_absh} a_1 a_2).
  \]

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

\[
\text{abs}((10100100) \text{ on } 0^{3600}(1) \text{ on } (1^{720}0^{720}1^{720}0^{720}1^{720}0^{720}1^{720}0^{720}1^{720}))
\]
\[
= \langle 0, \frac{7}{8} \rangle \text{ on } \langle -3600, -3600 \rangle (1) \text{ on } \langle -400, 480 \rangle (4/9) = \langle -2000, -\frac{20153}{18} \rangle (\frac{1}{6})
\]

minimal delay and buffer:

<table>
<thead>
<tr>
<th>delay</th>
<th>buffer size</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact result</td>
<td>9 598 (≈ time to receive 5 HD lines)</td>
</tr>
<tr>
<td>abstract result</td>
<td>11 995 (≈ time to receive 6 HD lines)</td>
</tr>
</tbody>
</table>

This is implemented in Lucy-N [http://lucy-n.org](http://lucy-n.org) by Louis Mandel.
Parallel implementation and integer clocks
Parallel processes communicating through a buffer

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

<table>
<thead>
<tr>
<th>Batch size</th>
<th>Cycles/push</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>23.07</td>
<td>589.45 MB/s</td>
</tr>
<tr>
<td>002</td>
<td>15.79</td>
<td>861.40 MB/s</td>
</tr>
<tr>
<td>004</td>
<td>12.06</td>
<td>1127.83 MB/s</td>
</tr>
<tr>
<td>008</td>
<td>10.00</td>
<td>1359.69 MB/s</td>
</tr>
<tr>
<td>016</td>
<td>7.51</td>
<td>1810.58 MB/s</td>
</tr>
<tr>
<td>032</td>
<td>7.33</td>
<td>1855.32 MB/s</td>
</tr>
<tr>
<td>064</td>
<td>7.33</td>
<td>1855.20 MB/s</td>
</tr>
</tbody>
</table>

**Batching**: reduce the synchronization with the FIFO
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!

Quite fruitful:

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


