Clocks in Kahn Process Networks

Marc Pouzet
École normale supérieure

L3, May 11, 2017
Dataflow Semantics

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.

<table>
<thead>
<tr>
<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>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</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>
</tbody>
</table>

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

A small kernel with minimal primitives

\[
e ::= e \text{ fby } e \mid \text{op}(e, \ldots, e) \mid x \mid i
\]

\[
\mid \text{merge } e \mid e \text{ when } e
\]

\[
\mid \lambda x.e \mid e \mid \text{rec } x.e
\]

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

| function ($\lambda x.e$), application ($e \; e$), fix-point (\text{rec } x.e) |
| constants $i$ and variables ($x$) |
| dataflow primitives : $x \text{ fby } y$ is the unitary delay; $\text{op}(e_1, \ldots, e_n)$ the point-wise application; sub-sampling/oversampling (when/merge). |
### Dataflow Primitives

<table>
<thead>
<tr>
<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>
</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>
</tr>
<tr>
<td>$x \text{ fby } y$</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>
</tr>
</tbody>
</table>

| $h$ | 1 | 0 | 1 | 0 | 1 | 0 |
| $x' = x \text{ when } h$ | $x_0$ | $x_2$ | $x_4$ |
| $z$ | $z_0$ | $z_1$ | $z_2$ |
| merge $h \ x' \ z$ | $x_0$ | $z_0$ | $x_2$ | $z_1$ | $x_4$ | $z_2$ |

### Sampling:

- if $h$ is a boolean sequence, $x \text{ when } h$ produces a sub-sequence of $x$
- merge $h \ x \ z$ combines two sub-sequences
Kahn Semantics

Every operator is interpreted as a stream function \((V^\infty = V^* + V^\omega)\). E.g., if \(x \mapsto s_1\) and \(y \mapsto s_2\) then the value of \(x + y\) is \(+\#(s_1, s_2)\)

\[
\begin{align*}
\# i & = i \# i \\
+\#(x.s_1, y.s_2) & = (x + y).+\#(s_1, s_2) \\
(x.s_1) \text{ fby} \# s_2 & = x.s_2 \\
x.s \text{ when} \# 1.c & = x.(s \text{ when} \# c) \\
x.s \text{ when} \# 0.c & = s \text{ when} \# c \\
\text{merge} \# 1.c x.s_1 s_2 & = x.\text{merge} \# c s_1 s_2 \\
\text{merge} \# 0.c s_1 y.s_2 & = y.\text{merge} \# c s_1 s_2
\end{align*}
\]
All this can be simulated in a few lines of Haskell

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)
After all, why do not use Haskell (or existing FP)?

We can write many useful examples and benefit from powerful type/module systems for free. Some of them are clearly real-time.

```haskell
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 (const 1)) in
  nat

-- resetable counter
reset_counter res input =
  let output = ifthenelse res (const 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)
               (const True)
    in o

filter n top = top when (every n)

hour_minute_second top =
    let second = filter (const 10) top in
    let minute = filter (const 60) second in
    let hour = filter (const 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 = (const 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)

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

$$u_1 = x$$

$\text{eps} = \text{const 0.001}$

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)
Some Programs generate monsters!

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

Structural (Scott) order:

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

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

- \( \text{first} \ (x:y) = x \)
- \( \text{next} \ (x:y) = y \)
- \( \text{incr} \ (x:y) = (x+1) : \text{incr} \ y \)
- \( \text{one} = 1 : \text{one} \)
- \( x = (\text{if } \text{hd}(\text{tl}(\text{tl}(\text{tl}(x)))) = 5 \text{ then } 3 \text{ else } 4) : 1 : 2 : 3 : \text{one} \)
- \( \text{output} = (\text{hd}(\text{tl}(\text{tl}(\text{tl}(x))))) : (\text{hd}(\text{tl}(\text{tl}(x)))) : (\text{hd}(x)) : \text{output} \)

The values are:
- \( x = 4 : 1 : 2 : 3 : 1 : ... \)
- \( \text{output} = 3 : 2 : 4 : 3 : 2 : 4 : ... \)
These stream may be constructed lazilly:

- \( x^0 = \bot, x^1 = \bot \cdot 1 : 2 : 3 : \text{un}, x^2 = 4 : 1 : 2 : 3 : \text{one}. \)
- \( \text{output}^0 = \bot, \text{output}^1 = 3 : 2 : 4 : \ldots \)

An other example (due to Paul Caspi):

\begin{verbatim}

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

\end{verbatim}

We have \( x = 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 10, 11, \ldots \).

Are they reasonnable programs? Streams are constructed in a reverse manner from the future to the past and 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:

\[ \text{half} = 0.\bot.0.\bot... \]

\[ \text{fail} = \text{fail} \]
\[ \text{half} = 0:\text{fail}:\text{half} \]
\[ \text{fill } x = (\text{hd}(x)) : \text{fill} (\text{tl}(\text{tl } x)) \]
\[ \text{ok} = \text{fill } \text{half} \]

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**:

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

**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 \( \bot \) in the Kahn semantics.
Kahn Semantics in Haskell

It is possible to remove possible non causal streams by forbidding values of the form \( \bot.x \). 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 :

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

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:**

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

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:

- \( (+) : int \times int \rightarrow int \) ← **its type signature**
- \( (+) :: \forall cl.cl \times cl \rightarrow cl \) ← **its clock signature**

In the following, we only consider the clock type.
Sampling

\[ s_1 \text{ when# } s_2 = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \]
\[ (\text{abs}.s) \text{ when# } (\text{abs}.c) = \text{abs}.s \text{ when# } c \]
\[ (v.s) \text{ when# } (1.c) = v.s \text{ when# } c \]
\[ (v.s) \text{ when# } (0.c) = \text{abs}.x \text{ when# } c \]

\[ \text{merge } c s_1 s_2 = \epsilon \text{ if one of the } s_i = \epsilon \]
\[ \text{merge } (\text{abs}.c)(\text{abs}.s_1)(\text{abs}.s_2) = \text{abs.merge } c s_1 s_2 \]
\[ \text{merge } (1.c)(v.s_1)(\text{abs}.s_2) = v.\text{merge } c s_1 s_2 \]
\[ \text{merge } (0.c)(\text{abs}.s_1)(v.s_2) = v.\text{merge } c s_1 s_2 \]
**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>$y = x \text{ when } h$</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 ...</td>
</tr>
<tr>
<td>$z = y \text{ when } h'$</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>$\text{merge } h' z k$</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 fby f in f
let node stutter x = o where
  rec o = merge five x ((0 fby o) whennot five) in o

stutter(nat) = 0.0.0.0.1.1.1.1.2.2.2.3.3...
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 $cl \text{ on } c$ : “$c$ moves at the pace of $cl$”

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

We get:

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

Written instead:

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

The previous program is now rejected.

This is a **typing error**

\[
\text{let } \text{even } x = x \text{ when half} \\
\text{let } \text{non}_{-}\text{synchronous } x = x \& (\text{even } x)
\]

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**
How to extend Lustre in a conservative way (without breaking it)?

**Build a “laboratory” language**

- a (quasi-dogmatic) attachment to the basic principles: stream Kahn semantics, clocks, functions
- study (implement) extensions of Lustre
- experiment things, manage all the compilation chain and write programs!

Quite fruitful:

- start of a close collaboration with the SCADE team at Esterel-Technologies
- the new SCADE 6 language (Oct. 2008) incorporates several features from Lucid Synchrone
- the LCM language at Dassault-Systèmes (Delmia Automation) based on the same principles
From Synchrony to Relaxed Synchrony

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

— 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 typical example: Picture in Picture

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

- **downscaler**: reduction of an HD image ($1920 \times 1080$ pixels) to an SD image ($720 \times 480$ 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
- Synchronous Kahn Networks

— based on the use of *infinite ultimately periodic sequences*

— a precedence relation $cl_1 <: cl_2$
Ultimately periodic sequences

$Q_2$ for the set of infinite periodic binary words.

$$(01) = 01 01 01 01 01 01 01 01 01 \ldots$$

$$0(1101) = 0 1101 1101 1101 1101 1101 1101 1101 \ldots$$

— 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

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

Sub-typing

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

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

sub-typing \[ w_1 \ll w_2 \iff \exists n \in \mathbb{N}, \forall i, 0 \leq \mathcal{O}(w_1)(i) - \mathcal{O}(w_2)(i) \leq n \]

synchronizability \[ w_1 \preceq w_2 \iff \exists b_1, b_2 \in \mathbb{Z}, \forall i, b_1 \leq \mathcal{O}(w_1)(i) - \mathcal{O}(w_2)(i) \leq b_2 \]

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

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

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

<table>
<thead>
<tr>
<th>base on ( p_1 )</th>
<th>( 1 1 0 1 0 1 0 1 0 1 \ldots )</th>
<th>( 1(10) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1 )</td>
<td>( 1 1 1 1 1 1 1 1 1 1 \ldots )</td>
<td>( 1(10) )</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>( 0 1 0 1 0 1 0 1 0 1 \ldots )</td>
<td>( 01 )</td>
</tr>
<tr>
<td>( (\text{base on } p_1) \text{ on } p_2 )</td>
<td>( 0 1 0 0 0 1 0 0 0 1 \ldots )</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

\[ H \vdash e_1 : ck \quad H \vdash e_2 : ck \]

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

Relaxed Synchrony:
- we add a sub-typing rule:

\[ H \vdash e : ck \text{ on } w \quad w <: w' \]

\[ (\text{SUB}) \quad H \vdash \text{buffer}(e) : ck \text{ on } w' \]

- 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.,
  \[(10100100) \text{ on } 0^{3600}(1) \text{ on } (101001001) = 0^{9600}(10^410^710^710^2)\]

- **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
     \[w = 00.(01 + (01))^*\]
     (e.g. \(w = 00\ 01\ 10\ 01\ 01\ 10\ 01\ 10\ 01\ 10\ldots\))

**Idea**: manipulate sets of clocks; turn questions into arithmetic ones
A word $w$ can be abstracted by two lines: $\text{abs}(w) = \langle b^0, b^1 \rangle (r)$

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

Instants

Number of ones

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

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

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

- finite number of state equivalence classes

- transition function \( \delta : \)
  \[
  \begin{align*}
  \delta(1,(i,j)) &= nf(i+1,j+1) \quad \text{if } j + 1 \leq r \times i + b^1 \\
  \delta(0,(i,j)) &= nf(i+1,j+0) \quad \text{if } j + 0 \geq r \times i + b^0
  \end{align*}
  \]

- 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: \( a_1 \prec: \sim a_2 \iff a_1 \bowtie \sim a_2 \land a_1 \preceq \sim a_2 \)

- proposition: \( \text{abs}(w_1) \prec: \sim \text{abs}(w_2) \Rightarrow w_1 \prec: w_2 \)
- buffer: \( \text{size}(a_1, a_2) = \lfloor b_1^1 - b_0^2 \rfloor \)
Abstract Operators

Composed clocks: \( c ::= w \mid \text{not} \ w \mid c \text{ on } c \)

Abstraction of a composed clock:

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

Operators correctness property:

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

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

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

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

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

**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 \( b^0_1 \leq 0, \quad b^0_2 \leq 0 \)
set of clock of rate $r = \frac{1}{3}$ and jitter 1 can be specified by $\langle -\frac{1}{3}, \frac{3}{3} \rangle \left( \frac{1}{3} \right)$

$\langle -\frac{1}{3}, \frac{3}{3} \rangle \left( \frac{1}{3} \right) = \langle -1, 1 \rangle \left( 1 \right) \text{ on } \sim \langle 0, \frac{2}{3} \rangle \left( \frac{1}{3} \right)$

$f :: \forall \alpha. \alpha \rightarrow \alpha \text{ on } \sim \langle -\frac{1}{3}, \frac{3}{3} \rangle \left( \frac{1}{3} \right)$
Most of the properties have been proved in Coq

- example of property

Property on_absh_correctness:

\[
\forall (w1:ibw) (w2:ibw), \forall (a1:abstractionh) (a2:abstractionh), \forall H_{wf_a1} : well_{formed_abstractionh} a1, \forall H_{wf_a2} : well_{formed_abstractionh} a2, \forall H_{a1_eq_absh_w1} : in_{abstractionh} w1 a1, \forall H_{a2_eq_absh_w2} : in_{abstractionh} w2 a2, \forall H_{in_abstractionh} (on w1 w2) (on_absh a1 a2).
\]

- number of Source Lines of Code

- specifications : about 1600 SLOC
- proofs : about 5000 SLOC
Back to the Picture in Picture Example

\[ \text{abs}((10100100) \text{ on } 0^{3600}(1) \text{ on } (1^{720}0^{720}1^{720}0^{720}1^{720}0^{720}1^{720}0^{720}1^{720})) = \langle 0, \frac{7}{8} \rangle \left( \frac{3}{8} \right) \text{ on } \langle -3600, -3600 \rangle (1) \text{ on } \langle -400, 480 \rangle \left( \frac{4}{9} \right) = \langle -2000, -\frac{20153}{18} \rangle \left( \frac{1}{6} \right) \]

- abstraction of downscaler output:
- minimal delay and buffer:

<table>
<thead>
<tr>
<th></th>
<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>
<td>192 240 (≈ 267 SD lines)</td>
</tr>
<tr>
<td>abstract result</td>
<td>11 995 (≈ time to receive 6 HD lines)</td>
<td>193 079 (≈ 268 SD 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

```c
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

<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
**Integer clocks**

\[ f \xrightarrow{\alpha \text{ on } (2)} g \]

**Burst:**
- allows to compute and communicate several values within one instant
- formulas can be easily lifted to integers
Burst:
- allows to compute several values into one instant
- formulas can be easily lifted to integers
- impacts causality

This is studied by Adrien Guatto in his PhD. thesis.
Références


