Synchrony and Clocks in Kahn Process Networks

Marc Pouzet
Univ. Paris-Sud 11
IUF

ISOR 2008, Algier
November 5, 2008

Joint work with Albert Cohen, Florence Plateau, Louis Mandel
Overview

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

Focus on systems which continuously interact with each others.

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

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

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

---

How can we program those systems, focusing first on the **functionality**, abstracting some implementation details?
The need for High-level Programming Languages

Conciliate three notions:

- a formal (and computable) model of time
  - express deadlines, simultaneous events, etc.

- parallelism to describe complex systems from simpler ones
  - control at the same time rolling and pitching
  - closed-loop systems (the controller and the plant run in parallel)

- statically guaranty safety properties
  - determinism, dead-lock freedom
  - execution in bounded time and memory

Safety is important:

- critical systems: fly-by-wire, braking, airbags, etc.

- properties must be guaranteed statically: “dynamic” = “too late”

- build the language on a strong mathematical basis to simplify verification/validation tasks
Synchronous Data-flow Languages

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

The idea of Lustre:

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

E.g, the linear filter defined by:

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

is programmed by writting the equation:

\[
Y = (0 \rightarrow a \ast \text{pre}(Y)) + b \ast X
\]

that is, we write **invariants**
An example of a SCADE specification
Kahn answered the following question: What is the semantics of a set of sequential processes communicating through FIFOs (e.g., Unix pipe, sockets)?

- message-based asynchronous communication (send/wait) through FIFOs
- reliable channels, bounded communication delays
- waiting on a single channel only. The program:
  
  if (A is present) or (B is present) then ...

is forbidden
Semantics

Domain:

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

Kahn Principle:

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

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

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

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

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

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

A natural model for video streaming applications (TV boxes): Sally (Philips NatLabs), StreamIt (MIT), Xstream (ST-micro) and restricted models \textit{à la SDF} (Ptolemy)
A Small Data-flow Kernel

Consider a small language kernel with basic data-flow primitives

\[ e ::= e \text{ fby } e \mid op(e, \ldots, e) \mid x \mid i \\
\mid \text{merge } e e e \mid e \text{ when } e \\
\mid \lambda x. e \mid e(e) \mid \text{rec } x.e \]

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

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

$h$ | 1 | 0 | 1 | 0 | 1 | 0  

$x$ when $h$ | $x_0$ | $x_2$ | $x_4$ |

$z$ | $z_0$ | $z_1$ | $z_2$ |

merge $h$ $x$ $z$ | $x_0$ | $z_0$ | $x_2$ | $z_1$ | $x_4$ | $z_3$ |

## Sampling:

- if $h$ is a boolean sequence, $x$ when $h$ produces a sub-sequence of $x$

- merge $h$ $x$ $z$ combines two sub-sequences
Kahn Semantics

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

\[
i\# = i.i\#
\]

\[
+\# (s_1, s_2) = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon
\]

\[
+\# (x.s_1, y.s_2) = (x + y).+\# (s_1, s_2)
\]

\[
\epsilon \text{ fb} y\# = \epsilon
\]

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

\[ \text{merge}^\# c s_1 s_2 = \epsilon \text{ if } s_i = \epsilon \]
\[ \text{merge}^\# 1.c x.s_1 s_2 = x.\text{merge}^\# c s_1 s_2 \]
\[ \text{merge}^\# 0.c s_1 y.s_2 = y.\text{merge}^\# c s_1 s_2 \]

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

**Derived operators:**

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

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

What happen when streams are sampled and composed together?

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

Execution with unbounded FIFOs!

Remarks:

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

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

\[
s : (V^{abs})^\infty
\]

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

\[
IB = \{0, 1\}
\]
\[
CLOCK = IB^\infty
\]
\[
clock(\epsilon) = \epsilon
\]
\[
clock(abs.x) = 0.c\text{lock } x
\]
\[
clock(v.x) = 1.c\text{lock } x
\]

Clocked Streams:

\[
ClStream(V, cl) = \{s | s \in (V^{abs})^\infty \land \text{clock } s \leq_p cl\}
\]
Data-flow Primitives

Constant generator:

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

Pointwise application:

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

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

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

Restrict the domain:

\((+): \forall cl : \text{CLOCK}. \text{ClStream}(\text{int}, cl) \times \text{ClStream}(\text{int}, cl) \to \text{ClStream}(\text{int}, cl)\)

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

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

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

\[
\begin{align*}
\text{• } (+) : \text{int} \times \text{int} & \to \text{int} & \text{← its type signature} \\
\text{• } (+) : \forall cl. cl \times cl & \to cl & \text{← its clock signature}
\end{align*}
\]

In the sequel, we only write the clock signature.
Delays

\[ \epsilon \text{fby}^\# s = \epsilon \]

\[ (\text{abs}.s_1) \text{fby}^\# (\text{abs}.s_2) = \text{abs}.(s_1 \text{fby}^\# s_2) \]

\[ (v.s_1) \text{fby}^\# (w.s_2) = v.(\text{fby}_1^\# w s_1 s_2) \]

\[ \text{fby}_1^\# v \in s = \epsilon \]

\[ \text{fby}_1^\# v (\text{abs}.s_1) (\text{abs}.s_2) = \text{abs}.(\text{fby}_1^\# v s_1 s_2) \]

\[ \text{fby}_1^\# v (w.s_1) (v'.s_2) = v.(\text{fby}_1^\# v' s_1 s_2) \]

As a consequence:

\[ \text{fby} : \forall cl.cl \times cl \rightarrow cl \]
Sampling

\[ s_1 \text{ when}^\# s_2 = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \]

\[ (\text{abs}. s) \text{ when}^\# (\text{abs}. c) = \text{abs}. s \text{ when}^\# c \]

\[ (v.s) \text{ when}^\# (1.c) = v.s \text{ when}^\# c \]

\[ (v.s) \text{ when}^\# (0.c) = \text{abs}. x \text{ when}^\# c \]

\[ \text{merge } c s_1 s_2 = \epsilon \text{ if one of the } s_i = \epsilon \]

\[ \text{merge } (\text{abs}. c) (\text{abs}. s_1) (\text{abs}. s_2) = \text{abs}. \text{merge } c s_1 s_2 \]

\[ \text{merge } (1.c) (v.s_1) (\text{abs}. s_2) = v.\text{merge } c s_1 s_2 \]

\[ \text{merge } (0.c) (\text{abs}. s_1) (v.s_2) = v.\text{merge } c s_1 s_2 \]
## 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 ...</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>
Sampling and Clocks

- in $x \text{ when} \# y$, $x$ and $y$ must have the same clock $cl$

- the clock of $x \text{ when} \# c$ is noted $cl$ on $c$: it means that $c$ moves at the pace $cl$

\[
s \text{ on } c = \epsilon \text{ if } s = \epsilon \text{ or } c = \epsilon
\]

\[
(1.cl) \text{ on } (1.c) = 1.cl \text{ on } c
\]

\[
(1.cl) \text{ on } (0.c) = 0.cl \text{ on } c
\]

\[
(0.cl) \text{ on } (\text{abs}.c) = 0.cl \text{ on } c
\]

We get:

\[\text{when} : \forall cl. \forall x : cl. \forall c : cl. cl \text{ on } c\]

\[\text{merge} : \forall cl. \forall c : cl. \forall x : cl \text{ on } c. \forall y : cl \text{ on } \text{not c}. cl\]

For any clock $cl$, if the first input $x$ has clock $cl$ and the second input $c$ has clock $cl$ then $x \text{ when } c$ has clock $cl$ on $c$. 
Checking Synchrony

The previous programs is now statically rejected by the compiler.

This is essentially a **typing problem**:

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

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

In synchronous languages, we only consider **clock equality**
From pure synchrony to $N$-synchrony

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

high definition (HD) → standard definition (SD)

1920 × 1080 pixels \( \rightarrow \) 720 × 480 pixels

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

**vertical filter:** number of lines from 1080 downto 480

---

**Real-Time Constraints**

the input and output processes: 30Hz.

HD pixels arrive at \( 30 \times 1920 \times 1080 = 62,208,000 \text{Hz} \)

SD pixels at \( 30 \times 720 \times 480 = 10,368,000 \text{Hz} \) (6 times slower)
But too restrictive for our video applications

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

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

- can we compute it automatically and obtain regular synchronous code?

we need a relaxed model of synchrony and relaxed clock calculus
**N-Synchronous Kahn Networks**

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

![Diagram showing the relationship between clocks](image)

- based on the use of *infinite ultimately periodic clocks*
- a precedence relation between clocks $c k_1 <: c k_2$
Infinite Ultimately Periodic Clocks

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

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

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

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

Definition:

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

Precedence relation: \( w_1 \preceq w_2 \)

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

Synchronisability:

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

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

Subsumption (sub-typing): \( w_1 <: w_2 \iff w_1 \blacktriangleright w_2 \land w_1 \preceq w_2 \)
Clocks represented graphically

Notations:

- $w[i]$ : element at index $i$
- $[w]_j$ : position of the $j^{th}$ 1
- $\mathcal{O}_w(i)$: number of 1s seen in $w$ until index $i$.

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

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

**synchronizability** $w_1 \asymp w_2 \iff \exists b_1, b_2 \in \mathbb{Z}, \forall i, b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$
Multi-sampled Systems (clock sampling)

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

\textit{c on w} denotes a \textit{subsampled clock}.

\textit{c on w} is the clock obtained in advancing in \textit{w} at the pace of clock \textit{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 1 1 ...</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{p}_1</td>
<td>1 1 0 1 0 1 0 1 0 1 0 1 ...</td>
<td>1(10)</td>
</tr>
<tr>
<td>base on \textit{p}_1</td>
<td>1 1 0 1 0 1 0 1 0 1 0 1 ...</td>
<td>1(10)</td>
</tr>
<tr>
<td>\textit{p}_2</td>
<td>0 1 0 1 0 1 0 1 0 1 0 1 ...</td>
<td>(01)</td>
</tr>
<tr>
<td>(base on \textit{p}_1) on \textit{p}_2</td>
<td>0 1 0 0 0 1 0 0 0 1 0 1 ...</td>
<td>(0100)</td>
</tr>
</tbody>
</table>
Computing with Periodic Clocks

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

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

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

**Delays and Buffers:** can be computed practically after normalisation.

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

\[
\begin{align*}
w & ::= u(v) \\
c & ::= w \mid c \text{ on } w \mid \text{not } c \mid \text{pre}(c) \mid \ldots
\end{align*}
\]
From Pure-Synchrony to \( N \)-Synchrony

Pure-Synchrony:

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

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

N-Synchrony:

- extend the basic clock calculus with a \textbf{sub-typing rule}:

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

- defines the synchronisation points where buffer code should be inserted
Going further: what about non periodic systems?

• Introducing clock relations gives more flexibility with as much guaranties as in synchronous model. No deadlocks, no buffer overflows.

• Subtyping relation can be checked provided clocks are periodic.

• Computing with exact period is unfeasible in practice. E.g.,

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

• Motivations:

  1. dealing with long patterns in periodic clocks. Avoid exact computation.

  2. specify/model jittering, i.e., how to deal with “almost periodic” clocks ? For instance

\[\alpha \text{ on } w \text{ with } w = 00.( (10) + (01) )*\]

\[(\text{e.g. } w = 00 \ 01 \ 10 \ 01 \ 01 \ 10 \ 01 \ 10 \ldots )\]

Idea: Manipulate sets of clocks instead of clocks. Transform the synchronisation problem into a linear problem with rational numbers.
Clock abstraction (work in progress)

\[ \text{concr} \left( (b^0, b^1, r) \right) \stackrel{\text{def}}{=} \]

\[
\left\{ \begin{array}{ll}
 w, \ \forall i \geq 1, & w[i] = 1 \implies O_w(i - 1) < r \times i + b^1 \\
 & w[i] = 0 \implies O_w(i - 1) \geq r \times i + b^0
\end{array} \right. 
\]

\[ w_1 : (0, 0, \frac{3}{5}) \text{ and } w_2 : (-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5}) \]
- Initial sets of 1s are well abstracted.
• Clocks with a null rate can be abstracted.
Properties

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

\[
early_a = \bigcap \{w, \forall i \geq 1, w[i] = 1 \implies O_w(i - 1) < r \times i + b^1\}
\]
\[
late_a = \bigcup \{w, \forall i \geq 1, w[i] = 0 \implies O_w(i - 1) \geq r \times i + b^0\}
\]

Proposition 1 (bounds of the enveloppe).
\[
\forall w \in \text{concr}(a), (early_a \preceq w) \land (w \preceq late_a).
\]

Proposition 2 (Empty concretisation). \(\forall a, \text{concr}(a) = \emptyset \iff early_a \not\preceq late_a\).

Proposition 3 (Early and Late binary words).
\[
\forall i,\quad O_{early_a}(i) = \max(0, \min(i, \lceil r \times i + b^1 \rceil))
\]
\[
O_{late_a}(i) = \max(0, \min(i, \lceil r \times i + b^0 \rceil))
\]

Proposition 4 (Non-empty enveloppe).
\[
\forall a = (b^0, b^1, r), b^0 \leq b^1 \implies \text{concr}(a) \neq \emptyset.
\]

Proposition 5 (Perfect Periodic Clock). \(|\text{concr}(b^0, b^1, r)| = 1\).
Clock enveloppes as circuits (i.e., automata)

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

\(<:\), \(\ast:\), etc. are the classical operation lifted to rational.

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

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

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

We only need integer arithmetic. In the same way, we can implement a generator, which either non-deterministically produce a clock within an enveloppe or the early or late bounds.
Abstract Operators: \( \text{not} \) and \( \text{on} \)

Replace exact computation with \( \text{not} \) and \( \text{on} \) by abstract ones.

\[
\text{not} \ ( ((b^0, b^1, r)) = (\neg b^1, \neg b^0, 1 - r)
\]

**Property:** \( a = \text{not} \ \text{not} \ a \)

If \( b^0_1 \leq 0 \) and \( b^0_2 \leq 0 \):

\[
(b^0_1, b^1_1, r_1) \ \text{on} \ (b^0_2, b^1_2, r_2) = (b^0_{12}, b^1_{12}, r_{12})
\]

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

**Abstraction of a sampled clock:**

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

\[
\text{abs}(\text{not} \ w) \overset{\text{def}}{\Leftrightarrow} \text{not} \ \text{abs}(w)
\]

\[
\text{abs}(c_1 \ \text{on} \ c_2) \overset{\text{def}}{\Leftrightarrow} \text{abs}(c_1) \ \text{on} \ \text{abs}(c_2)
\]

**Proposition:** Those operations are correct, i.e., \( c \in \text{concr}(\text{abs}(c)) \)
Abstract Relations: $\preccurlyeq \sim$, $\leq \sim$, $<$:~

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

$$(b^0_1, b^1_1, r_1) \preccurlyeq \sim (b^0_2, b^1_2, r_2) \iff r_1 = r_2$$

Proposition: $\text{abs}(c_1) \preccurlyeq \sim \text{abs}(c_2) \iff c_1 \preccurlyeq c_2$

Checking precedence is checking an arithmetic inequality

$$b^0_1 \geq b^1_2 \implies a_1 \leq \sim a_2$$

Proposition: $\text{abs}(c_1) \leq \sim \text{abs}(c_2) \Rightarrow c_1 \leq c_2$

$$a_1 <: \sim a_2 \iff a_1 \preccurlyeq \sim a_2 \land a_1 \leq \sim a_2$$

$\implies$ Subtyping can be checked in constant time.
Modelizing Execution Time

\[ f :: \forall \alpha. \alpha \text{ on} \sim (0, 0, \frac{1}{2}) \rightarrow \alpha \text{ on} \sim (-\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}) \text{ i.e.} \]
\[ \alpha \text{ on} \sim (0, 0, \frac{1}{2}) \rightarrow \alpha \text{ on} \sim (-3, -1, 1) \text{ on} \sim (0, 0, \frac{1}{2}) \]

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

Composed twice: \( f \circ f :: \forall \alpha. \alpha \text{ on} \sim (0, 0, \frac{1}{2}) \rightarrow \alpha \text{ on} \sim (-\frac{6}{2}, -\frac{2}{2}, \frac{1}{2}) \)
Modelizing Several Reads (or writes) at the Same Instant

![Graph showing the number of ones across instants]
Conclusion

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


