# *N*-Synchronous Kahn Networks
## A Relaxed Model of Synchrony for Real-Time Systems

## A Domain-Specific Program Generation Approach

*or* Topiarization is Better Than Deforestation
*or* Topiary Rather Than Listless Evaluation

**Albert Cohen[1], Marc Duranton[2], Christine Eisenbeis[1], Claire Pagetti[1,4], Florence Plateau[3] and Marc Pouzet[3]**

IFIP WG2.11

1: INRIA, Orsay, France
2: PHILIPS NatLabs, Eindhoven, The Netherlands
3: University of Paris-Sud, Orsay, France
4: ONERA, Toulouse, France

# Context

- **Video intensive applications** (TV boxes, medical systems)
  *tera-operations per second (on pixel components) is typical*
- **Ensure three properties: hard real-time and performance and safety**

## Implementations

- Today: specific hardware (ASIC)
- Evolution: multi-clock asynchronous architectures, mixing hardware/software because of costs, variability of supported algorithms

## Design and programming tools

- General purpose languages and compilers are not well adapted
- Kahn Networks (KN) is common practice in the field

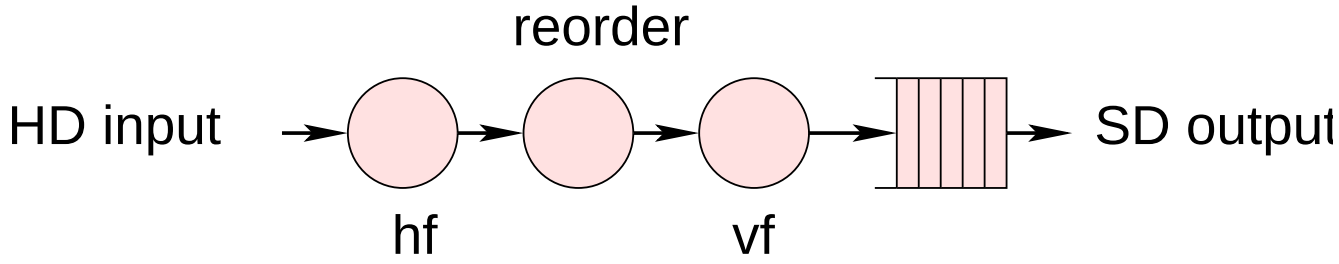
# A typical example: the Downscaler

High definition (HD) $\rightarrow$ Standard definition (SD)

$1920 \times 1080$ pixels $720 \times 480$

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

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




## Real-Time Constraints

Input and output processes: 30Hz.

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

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

# Our Goal

Define a **programming language** dedicated to those Kahn Networks providing:

- a modular functional description
- a modular description of the timing requirements

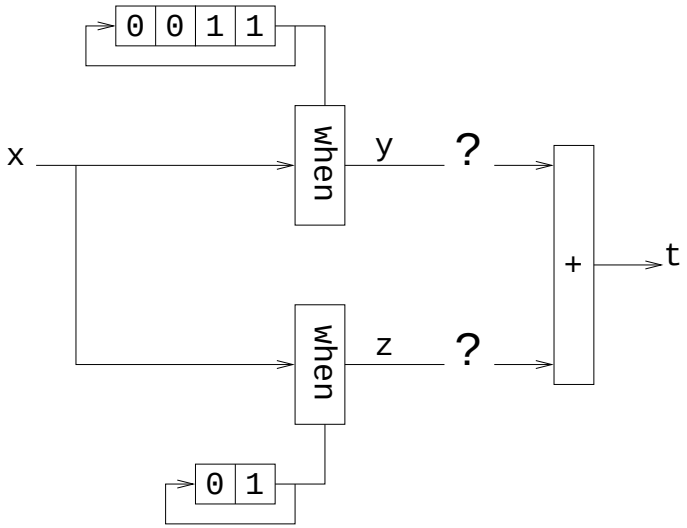
with a semantics and a compiler which **statically guarantees** four important properties. E.g., on the downscaler:

- a proof that, according to worst-case time conditions, the frame and pixel rate will be sustained
- a proof that the system executes in bounded memory
- an evaluation of the delay introduced by the downscaler in the video processing chain, i.e., the delay before the output process starts receiving pixels
- an evaluation of memory requirements, to store data within the processes, and to buffer the stream produced by the vertical filter in front of the output process

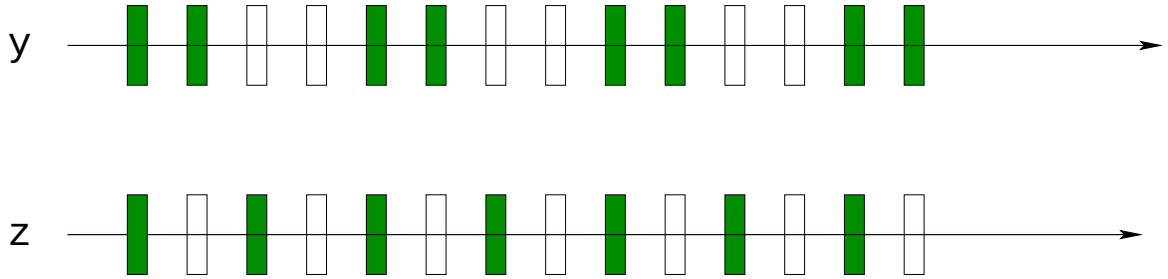
## What about Synchronous Languages?

- Dedicated to hard real-time critical systems
- Static analysis, verification and testing tools
- Synchrony is ensured by a type-system for clocks: a **clock calculus**
- It allows to program **Synchronous Kahn Networks**
- **Type-directed generation** of custom hardware/software system with static guarantees (real-time and resource constraints)

# But too restrictive for our video applications




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

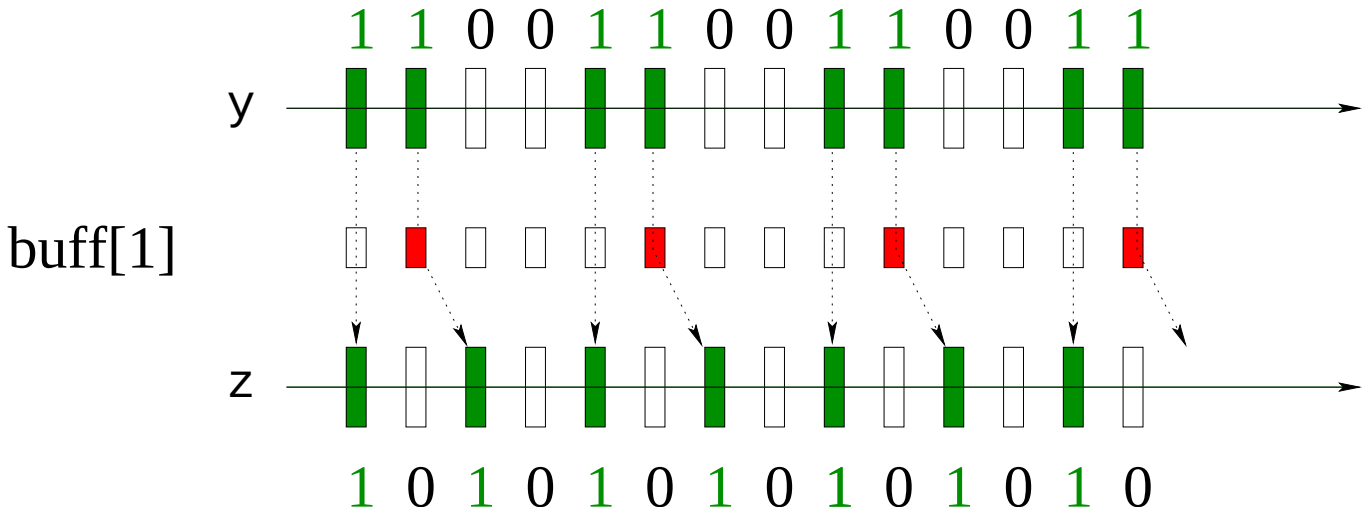



- Adding buffer code (by hand) is feasible but hard and error-prone
- Can we compute it automatically and obtain regular synchronous code?

**We need a relaxed model of synchrony and relaxed clock calculus**

# $N$-Synchronous Kahn Networks

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




- Based on the use of *infinite ultimately periodic clocks*
- A precedence relation between clocks $ck_1 <: ck_2$

# Infinite Ultimately Periodic Clocks

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

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

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

**Definition:**

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

# Causality order and Synchronisability

**Precedence relation:** $w_1 \preceq w_2$

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

**Synchronisability:**

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

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

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

# Multi-sampled Systems (clock sampling)

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

$c$ on $w$ denotes a *subsampled clock*.

$c$ on $w$ is the clock obtained in advancing in $w$ at the pace of clock $c$. 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) |

**Proposition 1 (on-associativity)** *Let $w_1$, $w_2$ and $w_3$ be three infinite binary words. Then $w_1$ on $(w_2$ on $w_3) = (w_1$ on $w_2)$ on $w_3$.*

# Computing with Periodic Clocks

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

**Synchronizability:** Two infinite periodic binary words $u(v)$ and $u'(v')$ are *synchronizable*, noted $u(v) \bowtie u'(v')$ iff they have the same rate, i.e., $\frac{|v|_1}{|v'|_1} = \frac{|v|}{|v'|}$.

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

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

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

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

# A Synchronous Data-flow Kernel

- Reminiscent to `Lustre` and `Lucid Synchrone`
- receive a standard (strictly) synchronous semantics

$$
\begin{array}{lcl}
e & ::= & x \mid i \mid e \texttt{ where } x = e \mid e(e) \\
 & & \quad \mid e \texttt{ fby } e \mid e \texttt{ when } pe \mid \texttt{merge } pe\ e\ e \\
 & & \\
d & ::= & \texttt{let node } f(x) = e \mid d; d \\
 & & \\
dp & ::= & \texttt{period } p = pe \mid dp; dp \\
 & & \\
pe & ::= & p \mid w \mid pe \texttt{ on } pe \mid \texttt{not } pe \mid \ \ldots
\end{array}
$$

- `fby` is the initialized delay (or *register*)
- `when` is the *sampling* operator allowing to extract a sub-stream from a stream
- `merge` is the *combination* operator allowing to combine two complementary streams (with opposite clocks)

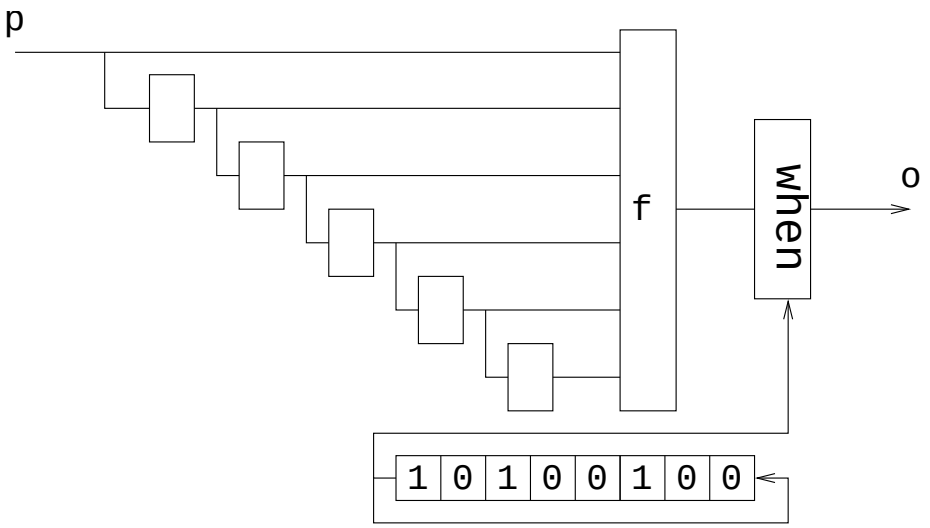
# The Downscaler

```
let period c = (10100100)

let node hf p = o where
  rec o2 = 0 fby p
  and o3 = 0 fby o2
  and o4 = 0 fby o3
  and o5 = 0 fby o4
  and o6 = 0 fby o5
  and o = f (p,o2,o3,o4,o5,o6) when c
```

```
val hf : 'a -> 'a on c
```

```
let node main i = o where
  rec t = hf i
  and (i1,i2,i3,i4,i5,i6) = reorder t
  and o = vf (i1,i2,i3,i4,i5,i6)
```




- The clock signature of each process abstracts its timing behavior
- Clock calculus: what is the clock signature of `main`?

# Clock calculus

$$
\begin{array}{lcl}
\sigma & ::= & \forall\alpha.\sigma \mid ct \\
ct & ::= & ct \rightarrow ct \mid ct \times ct \mid ck \\
ck & ::= & ck \ \texttt{on} \ pe \mid \alpha \\
H & ::= & [x_1 : \sigma_1, \ldots, x_m : \sigma_m] \\
P & ::= & [p_1 : w_1, \ldots, p_n : w_n]
\end{array}
$$

**Judgment:** $P, H \vdash e : ct$ "expression $e$ receive clock type $ct$ in environments $H$ and $P$"

# From 0-Synchrony to $N$-Synchrony

**0-Synchrony:**

- 0-synchrony can be checked using standard Milner-type system [ICFP'96,Emsoft'03]
- only need clock equality (and clocks are not necessarily periodic)

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

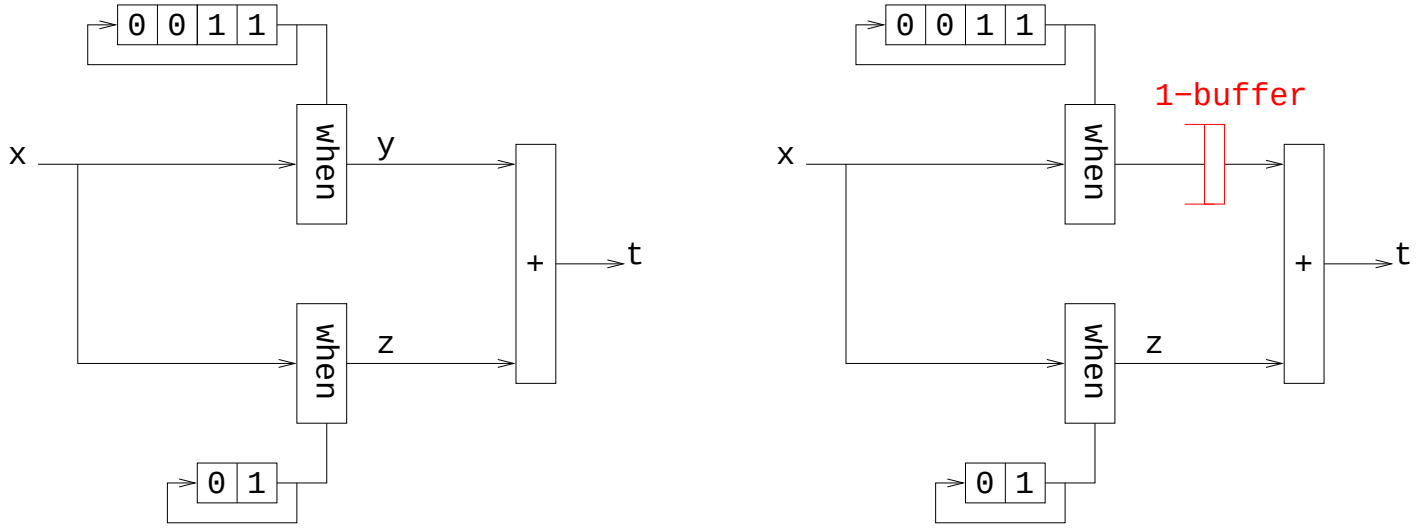
**N-Synchrony:**

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

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

- defines the synchronisation points where buffer code should be inserted

# An Example




$$\texttt{let node } f(x) = t \texttt{ where } t = (x \texttt{ when } (1100)) + (x \texttt{ when } (10))$$

(1100) and (10) can be synchronized using a buffer of size 1. Indeed:

$$\frac{P, H \vdash x \texttt{ when } (1100)) : \alpha \texttt{ on } (1100) \quad (1100) <: (10)}{P, H \vdash x \texttt{ when } (1100) : \alpha \texttt{ on } (10)}$$

Finaly, $f : \forall\alpha.\alpha \to \alpha \texttt{ on } (01)$
and the 1-buffer `buffer[1]`: $\forall\alpha.\alpha \texttt{ on } (1100) \to \alpha \texttt{ on } (1010)$

# Translation into 0-Synchronous Programs

$$\text{(TRANSLATION)}\ \frac{P, H \vdash e : ck\ \texttt{on}\ w \Rightarrow e' \quad w <: w'}{P, H \vdash e : ck\ \texttt{on}\ w' \Rightarrow \texttt{buffer}_{w,w'}(e')}$$

$$\texttt{buffer}_{w,w'} : \forall\alpha.\alpha\ \texttt{on}\ w \rightarrow \alpha\ \texttt{on}\ w'$$

**Theorem (correctness):** Any well clocked ($N$-synchronous) program can be transformed into a 0-synchronous program

This is a constructive proof: sub-typing points define where some buffering is necessary

The translated program can be checked with the basic clock calculus

# Algorithm (constraint resolution)

The sub-typing system is not deterministic and is thus not an algorithm

**Standard solution:**

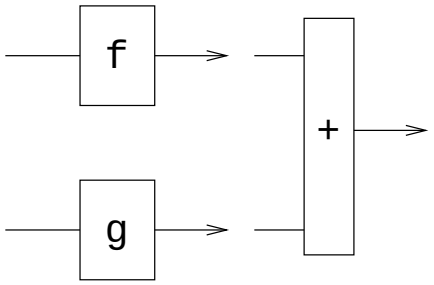
- apply the (SUB) rule at every program construction
- generate a set of sub-typing constraints $\{ck_1 <: ck'_1, \ldots, ck_n <: ck'_n\}$
- rely on a resolution algorithm

Resolution amounts to rewriting (simplifying) the set of constraints until we get the empty set

**Theorem (completeness):** For any expression $e$, and for any period and clock environments $P$ and $H$, if $e$ has an admissible clock type in $P, H$ for the relaxed clock calculus, then the type inference algorithm computes a clock $ct$ verifying $P, H \vdash e : ct$

# Clock sampling (gating) *vs* Buffering

In general, there exists an infinite number of solutions.




$$f \quad : \quad \forall \alpha_1.\alpha_1 \rightarrow \alpha_1 \ \texttt{on} \ (1100)$$
$$g \quad : \quad \forall \alpha_2.\alpha_2 \rightarrow \alpha_2 \ \texttt{on} \ (10)$$
$$(+) \quad : \quad \forall \alpha_3.\alpha_3 \times \alpha_3 \rightarrow \alpha_3$$

We have to solve the constraint: $\alpha_1 \ \texttt{on} \ (1100) <: \alpha_3$ and $\alpha_2 \ \texttt{on} \ (10) <: \alpha_3$

**Clock sampling: (unification)**

find $v_1$ and $v_2$ st $\alpha_1 = \alpha_4 \ \texttt{on} \ v_1$ and $\alpha_2 = \alpha_4 \ \texttt{on} \ v_2$

Solution: $\alpha_4 \ \texttt{on} \ (10111) \ \texttt{on} \ (1100) = \alpha_4 \ \texttt{on} \ (10100) = \alpha_3$

$\alpha_4 \ \texttt{on} \ (11110) \ \texttt{on} \ (10) = \alpha_4 \ \texttt{on} \ (10100) = \alpha_3$

**No buffering but the base clock must be faster**

**Buffering: (sub-typing)**

$\alpha_1 = \alpha_4$ and $\alpha_2 = \alpha_4$, $\alpha_4 \ \texttt{on} \ (1100) <: \alpha_3$ and $\alpha_4 \ \texttt{on} \ (10) <: \alpha_3$

$\alpha_4 \ \texttt{on} \ (1100) \sqcup (10) = \alpha_4 \ \texttt{on} \ (10) = \alpha_3$

**A 1-buffer is needed**

(EQUAL)
$$S \rightsquigarrow S\left[\begin{array}{c}\alpha_1' \texttt{ on } \mathcal{V}(w_1,w_2)/\alpha_1 \\ \alpha_2' \texttt{ on } \mathcal{V}'(w_1,w_2)/\alpha_2\end{array}\right]$$
$$\text{if } S = S' + I_1 + I_2, \quad \begin{array}{ll} I_1 = \{\alpha_1 \texttt{ on } w_1 <: ck_1\} \text{ or } \{ck_1 <: \alpha_1 \texttt{ on } w_1\} & \alpha_1 \neq \alpha_2 \\ I_2 = \{\alpha_2 \texttt{ on } w_2 <: ck_2\} \text{ or } \{ck_2 <: \alpha_2 \texttt{ on } w_2\} & w_1 \neq w_2 \end{array},$$

(CYCLE)
$$S + \{\alpha \texttt{ on } w_1 <: \alpha \texttt{ on } w_2\} \rightsquigarrow S$$
$$\text{if } w_1 <: w_2$$

(SUP)
$$S + \{\alpha \texttt{ on } w_1 <: \alpha',\ \alpha \texttt{ on } w_2 <: \alpha'\} \rightsquigarrow S + \{\alpha \texttt{ on } w_1 \sqcup w_2 <: \alpha'\}$$
$$\text{if } w_1 \bowtie w_2$$

(INF)
$$S + \{\alpha' <: \alpha \texttt{ on } w_1,\ \alpha' <: \alpha \texttt{ on } w_2\} \rightsquigarrow S + \{\alpha' <: \alpha \texttt{ on } w_1 \sqcap w_2\}$$
$$\text{if } w_1 \bowtie w_2$$

(CUT)
$$S + \{\alpha_1 \texttt{ on } w <: \alpha_2 \texttt{ on } w\} \rightsquigarrow S + \{\alpha_1 <: \alpha_3 \texttt{ on } u_1,\ \alpha_3 \texttt{ on } u_2 <: \alpha_2\}$$
$$\text{if } \alpha_1 \neq \alpha_2,\ u_1 = \mathcal{U}_{\max}(w),\ u_2 = \mathcal{U}_{\min}(w)$$

(FORK)
$$S + \{\alpha <: \alpha_1 \texttt{ on } w,\ \alpha <: \alpha_2 \texttt{ on } w\} \rightsquigarrow S[\alpha_3 \texttt{ on } u \texttt{ on } w/\alpha] + \{\alpha_3 \texttt{ on } u <: \alpha_1,\ \alpha_3 \texttt{ on } u <: \alpha_2\}$$
$$\text{if } \alpha_1 \neq \alpha_2,\ u = \mathcal{U}_{\min}(w)$$

(JOIN)
$$S + \{\alpha_1 \texttt{ on } w <: \alpha,\ \alpha_2 \texttt{ on } w <: \alpha\} \rightsquigarrow S[\alpha_3 \texttt{ on } u \texttt{ on } w/\alpha] + \{\alpha_1 <: \alpha_3 \texttt{ on } u,\ \alpha_2 <: \alpha_3 \texttt{ on } u\}$$
$$\text{if } \alpha_1 \neq \alpha_2,\ u = \mathcal{U}_{\max}(w)$$

(SUBST)
$$S \oplus I \rightsquigarrow S[ck/\alpha]$$
$$\text{if } I = \{\alpha <: ck\} \text{ or } \{ck <: \alpha\},\ \alpha \notin FV(ck)$$

# Conclusion

- $N$**-Synchronous Kahn Networks** introduce a relaxed model of synchrony
- extended synchrononous framework: automatic generation of the synchronous buffers which are semantically (as defined by Kahn) guaranteed correct
- a relaxed clock calculus where buffering corresponds to sub-typing
- $N$-synchronous programs can be translated into 0-synchronous ones
- extend the expressive power of synchronous languages, yet allowing to do compilation, simulation and verification after translation
- **Lustre programs are 0-Synchronous Kahn Networks**
- **Kahn networks are $\infty$-Synchronous Kahn Networks**

# Current and Future Work

- algebraic characterisation and symbolic representation of clocks
- implementation inside an existing synchronous language
- optimization and architecture considerations (buffer size, locality, clock gating)
- forgetting buffering mechanism, periodic clocks are useful for dealing with several implementations of the same function:
  - parallel *vs* pipelined *vs* serial computation
  - going from a version to an other changes clocks
  - how to prove them to be equivalent (or to derive them from the same program) according to resource constraints?