Type-based Clock Calculi

Marc Pouzet

ENS Paris
Marc.Pouzet@ens.fr

MPRI, October 2024

Different interpretations of the words “synchronous” and “clocks” exist.

Kahn Process Networks

A set of processes running independently and synchronizing through FIFOs [5].

In this context, what does it mean for two streams to be "synchronous" or two processes to run "synchronously"?

- The Synchronous Data-flow Model (SDF) model of Messerschmitt & Lee [6] defines relative ratio between input reads and output writes of a process.
- E.g., "Every time f read 2 input on channel x and 3 inputs on channel y, it produces 5 outputs on channel z".
- Express constraints of relative input reads and write as balanced equations.
- A SDF network is correct (alive = deadlock free and bounded) when balanced equations have a solution.
- It ensures that a static schedule exists and a run with bounded buffer whose maximum size can be computed [7].
- Since [6], various SDF extensions have been considered.

Periodically sampled systems

Consider that parallel processes run in lock-step — when one process make a step, the others makes a step.

There exist a global time scale shared by all processes, e.g. 1ms.

"synchrony" can be interpreted as:

> *Two periodically sampled signals can be synchronized on the gcd of their period.*

E.g., if x is sampled every 10ms and y every 15ms, the signal $z = x + y$ needs to be computed every 5ms. z is sampled every 5ms.

In both interpretations, "synchrony" is a matter of relative speed (or rate) of production or change of value.

Synchronous Kahn Networks

synchrony of synchronous languages has a more general and powerful meaning.

That is, the two previous situations are particular cases.

> *A clock is a set of totally ordered instants. There exist a global clock so that every signal is defined according to this clock.*

A Kahn network is synchronous if it can be executed without any buffering mechanism.

Example: Synchronous Circuits

A synchronous circuit behave as if it all operator were running in lock-step, reading one input, producing one output.

In a synchronous circuit, it is possible to mimic the absence of a value by adding an enable bit: every wire s is paired with a boolean b, true when s is valid.

What is a Clock

The clock of a signal defines the instants where the signal is defined. It is its time domain as opposed to the domain of values.

If D is a set of instants, equation $z = x + y$ means:

$$\forall t \in D.z(t) = x(t) + y(t)$$

D is equiped with a total order. If D is a set of instants, or clock, and $D' \subseteq D$, D' is called a sub-clock of D.

Some operators can read or produce values at a subset of instants.

E.g., `x when c` returns a signal which is defined on a subclock $D' \subset D$ if `x` and `c` have clock D. It produces a signal which is defined on time $t \in D'$ if $x(t)$ is defined, $c(t)$ is defined and is true.

whereas `merge` takes two signals defined at complementary instants.

What is a Clock

The clock calculus is a type system on these time domains. It associates a **clock type** to an expression e which indicates when e produces a value.

When the program type checks, it can be executed synchronously without any buffering.

- Clocks are useful to mix slow and fast processes;
- while ensuring the absence of buffering.

From the programming language point-of-view:

- Clocks help specifying and reasonning about reactive processes.
- Clocks are useful to generate good code.
- In a way similar to types in programming languages.
  - A strong constraint on the programmer but increases safety.
  - It helps undestanding what the program is doing.

Clocks, in practice

The problem is not “easy” in the general case. E.g.,:

$$(e_1 \texttt{ when } c_1) + (e_2 \texttt{ when } c_2)$$

is synchronous iff c_1 and c_2 are equal. If the language contains boolean expressions, it is *NP*-complete. If it contains boolean expressions with registers, it is *PSPACE*-complete. If it contains unbounded arithmetics, it is undecidable. In practice:

- Give sufficient conditions to insure that a program can be executed synchronously.
- Clock equality: structural equality (**Lucid Synchrone**); boolean equivalence (**Signal**).
- Clock inference: **Signal**, **Lucid Synchrone**.
- Clock verification: **Lustre**.

Remark:

This is a very general problem and tools like Simulink also provides a static checking of rates/clocks of block diagrams.

A small stream language

$$\begin{array}{rcl} f, e & ::= & e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid x \mid i \\ & & \mid e\ \texttt{fby}\ e \mid \texttt{merge}\ e\ e\ e \\ & & \mid e\ \texttt{->}\ e \mid e\ \texttt{when}\ e \\ & & \mid \texttt{rec}\ x.e \mid \lambda x.e \end{array}$$

- Streams and stream function.
- Regular typing is not addressed here, causality neither.
- Check only the operations are executed on their proper clock.

Finite and Infinite Streams

Let T be a set of value.

- If $n \in \mathbb{N}$, T^n is the set of sequences of length n.
- If $x \in T^n$, and $1 \leq i \leq n$, $x(i)$ is the i-th element of T^n. It is not defined otherwise.
- If $v \in T$ and $s \in T^n$, $v.s \in T^{n+1}$ is the stream with $(v.s)(1) = v$ and $(v.s)(i) = s(i-1)$, for $1 \leq i \leq n+1$.
- T^0 contains the empty sequence noted ϵ.
- The Kleene set $T^* = \cup_{n \in \mathbb{N}} T^n$ is the set of finite sequences.
- $T^\omega = lim_{n \to \infty} T^n$ is the set of infinite sequences.
- The set of finite and infinite streams is:

$$T^\infty = T^* \cup T^\omega$$

The Prefix Order

The binary relation $\leq_p \subseteq T^\infty \times T^\infty$ is the smallest such that:

- For all $s \in T^\infty, \epsilon \leq_p s$.
- For all s_1, s_2, if $s_1 \leq_p s_2$ then forall $v \in T$, $v.s_1 \leq_p v.s_2$

Clocked Streams

Let $T_{abs} = T + \{abs\}$, the set T complemented with a "absent" value.

Clocks

Let $x \in T_{abs}^{\infty}$. The clock $Clock(x) \in Bool^{\infty}$ of x is a boolean stream:

$$\begin{array}{lcl} Clock(\epsilon) & = & \epsilon \\ Clock(abs.s) & = & \texttt{false}.Clock(s) \\ Clock(v.s) & = & \texttt{true}.Clock(s) \end{array}$$

Clocked Stream

The set of clocked streams whose clock is s is defined:

$$ClockedStream(T, c) = \{s \in T^{\infty} \mid Clock(s) \leq_p c\}$$

$s \in ClockedStream(T, C)$ means
$\forall i \in Dom(s), (s(i) = abs) \Leftrightarrow (c(i) = \texttt{false})$

The set is prefix closed, i.e., if c is of length n, we allow $ClockedStream(T, c)$ to contains all shorter streams.

Static Checking

Intuition: associate a type to every expression. For a stream expression e, this type is interpreted as a boolean expression s whose value if true when e produce a present value.

The clock type language:

$$
\begin{array}{lcl}
\sigma & ::= & \forall \alpha_1, ..., \alpha_n . cl \\
cl & ::= & \forall x : cl.cl \mid cl \times cl \mid s \\
s & ::= & s \ \texttt{on} \ e \mid \alpha \\
 & & \\
H & ::= & [x_1 : \sigma_1, ..., x_n : \sigma_n] \\
 & & \text{where for all } i, j \text{ st } j \leq i, \ x_i \notin FV(cl_j) \\
 & & \\
\text{judgment} & ::= & H \vdash e : cl
\end{array}
$$

Programs are considered modulo α-conversion (renaming)

- A dependent type system.
- $\forall x : cl_1.cl_2$ is written $cl_1 \rightarrow cl_2$ when $x \notin FV(cl_2)$

Initial Conditions

$$
\begin{array}{lll}
H_0 & = & [\texttt{pre} \quad : \forall\alpha.\alpha \to \alpha, \\
 & & \;\texttt{->} \quad : \forall\alpha.\alpha \to \alpha \to \alpha, \\
 & & \;\texttt{when} \quad : \forall\alpha.\alpha \to \forall x : \alpha.\alpha \texttt{ on } x \\
 & & \texttt{merge} \quad : \forall\alpha.\forall x : \alpha.\alpha \texttt{ on } x \to \alpha \texttt{ on not } x \to \alpha
\end{array}
$$

Instantiation, generalisation:

- Free clock variables: $FV(cl)$. Lifted to environments: $FV(H)$.
- Free expression variables: $fv(cl)$. Lifted to environments: $fv(H)$.

$$cl[s_1/\alpha_1, ..., s_n/\alpha_n] \quad \in \quad Instanciate(\forall \alpha_1, ..., \alpha_n.cl)$$

$$\begin{array}{rcl} Generalize(H, cl) & = & \forall \alpha_1, ..., \alpha_m.cl \\ & & \text{where } \{\alpha_1, ..., \alpha_n\} = FV(cl) \backslash FV(H) \end{array}$$

Polymorphism is limited: a clock variable can be instantiated by a clock type s which concerns signals only.

The system

(Const)

$$H \vdash i : s$$

(Var)

$$\frac{cl \in Instanciate(\sigma)}{H, x : \sigma \vdash x : cl}$$

(op)

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

(abst)

$$\frac{H, x : cl \vdash e : cl' \qquad x \notin fv(H)}{H \vdash \lambda x.e : \forall x : cl.cl'}$$

(app)

$$\frac{H \vdash f : \forall x : cl.cl' \qquad H \vdash e : cl}{H \vdash f\ e : cl'[e/x]}$$

(REC)

$$\frac{H, x : cl \vdash e : cl \qquad x \notin fv(H)}{H \vdash \texttt{rec}\ x.e : cl}$$

(LET)

$$\frac{H \vdash e_1 : cl_1 \qquad H, x : Generalize(H, cl_1) \vdash e_2 : cl_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : cl_2}$$

Pairs

(FST)

$$H \vdash \texttt{fst} : \forall\alpha_1, \alpha_2.\alpha_1 \times \alpha_2 \to \alpha_1$$

(SND)

$$H \vdash \texttt{snd} : \forall\alpha_1, \alpha_2.\alpha_1 \times \alpha_2 \to \alpha_2$$

(PRODUCT)

$$\frac{H \vdash e_1 : cl_1 \qquad H \vdash e_2 : cl_2}{H \vdash (e_1, e_2) : cl_1 \times cl_2}$$

Polymorphism

- Polymorphism is limited: `fst` takes two streams and returns a stream since α denotes a variable which can only be instantiated by a clock expression of the form s on e.
- Pairs can be treated in a more general manner by extending the type language.

$$\begin{array}{lcl} \sigma & ::= & \forall \beta_1, ..., \beta_n . \forall \alpha_1, ..., \alpha_n . cl \\ cl & ::= & \forall x : cl.cl \mid cl \times cl \mid s \mid \beta \\ s & ::= & s \texttt{ on } e \mid \alpha \end{array}$$

- Then, `fst` and `snd` get clock signatures:

$$\text{(FST)}\quad H \vdash \texttt{fst} : \forall \beta_1, \beta_2 . \beta_1 \times \beta_2 \rightarrow \beta_1$$

$$\text{(SND)}\quad H \vdash \texttt{snd} : \forall \beta_1, \beta_2 . \beta_1 \times \beta_2 \rightarrow \beta_2$$

Polymorphism

An alternative solution is to keep a simpler clock type language.

$$\begin{array}{lcl} \sigma & ::= & \forall \beta_1, ..., \beta_n.cl \\ cl & ::= & \forall x : cl.cl \mid cl \times cl \mid \beta \mid cl \texttt{ on } e \end{array}$$

Yet, the meaning of some combinations must be defined (and is, at least unclear). E.g.,

- $(cl_1 \times cl_2)$ `on` e;
- $(\forall x : cl_1.cl_2)$ `on` e;
- ...

These situations can be rejected by the regular type system or taken into account by merging the type system and the clock calculus.

Extension: clock abstraction

How can we write a function (node) that returns a stream sampled on a condition c computed locally?
In Lustre, the condition must be returned as an output of the function.

```
node hide(x: int) returns (o: bool; (y:int) when o);
  let o = x >= 0;
      y = x when o;
  tel;
```

This corresponds to an existential quantification:

$$\texttt{hide} : \forall\alpha.\alpha \to \Sigma(o : \alpha).\alpha \texttt{ on } o$$

(RETURN)

$$\frac{H \vdash e_1 : cl_1 \qquad H \vdash e : cl_2[e_1/x]}{H \vdash (e_1, e_2) : \Sigma(x : cl_1).cl_2}$$

(FST)

$$\frac{H \vdash e : \Sigma(x : cl_1).cl_2}{H \vdash \texttt{fst}\ e : cl_1}$$

(SND)

$$\frac{H \vdash e : \Sigma(x : cl_1).cl_2}{H \vdash \texttt{snd}\ e : cl_2[\texttt{fst}\ e/x]}$$

The Valued Signals of Esterel

The language Esterel provides pure and valued signals. A pure signal is nothing but a boolean. A valued signal carries both a value and a presence bit. Using clocks, it can be encoded by a dependent pair:

$$\alpha\ \mathtt{sig} = \Sigma(c : \alpha).\alpha\ \mathtt{on}\ c$$

made of:

- An enable bit c;
- and a stream present when c is true.

Add two operations: one to abstract the enable bit; one to open it.

Clock abstraction

The equation:

$$\mathtt{emit}\ x = e$$

defines the valued signal x by abstracting the clock of e.

(Emit)

$$\frac{H \vdash e : s\ \mathtt{on}\ c}{H \vdash \mathtt{emit}\ x = e : [x : s\ \mathtt{sig}]}$$

Accessing an abstract clock

$\texttt{let}\ x\ \texttt{on}\ c = e_1\ \texttt{in}\ e_2$ access the signal e_1.

$$\textsc{(Let-sig)}\quad \frac{\begin{array}{c} c \notin fv(H) \qquad c \notin fv(cl) \\ H \vdash e_1 : s\ \texttt{sig} \qquad H, c : s, x : s\ \texttt{on}\ c \vdash e_2 : cl \end{array}}{H \vdash \texttt{let}\ x\ \texttt{on}\ c = e_1\ \texttt{in}\ e_2 : cl}$$

The rule ensures that no hypothesis on c can be made and it must not escape from the block.

Historical note: The idea of "clocks as (dependent) types" was introduced xsin ICFP'96 (Caspi & Pouzet). It was implemented in Lucid Synchrone V1 (1996-1998).

Oversampling

In **Lucid Synchrone**, Version 1.0 (1998), it was possible to write an oversampling function whose input clock could depend only its output, provided there was no instantaneous dependence.
E.g., take `f` and `terminated` two length preserving functions.

```
let node oversampling(x) = ok, o where
  rec cx = merge (true fby ok) x
                 ((0 fby cx) when not (true fby ok))
  and o = f(cx)
  and ok = terminated(o)
```

```
val oversampling :: 'a on true fby ok -> (ok: 'a) * 'a
```

This program mimics an internal loop that reads an input from time to time but produce an output at every instant.

Oversampling

The type $\Pi x : cl_1.cl_2$ expresses that the clock of output depends on the value of the input.

The type $\Sigma x : cl_1.cl_2$ expresses that the clock of the output depends on the value of the first component of the pair.

How to express the clock signature of a function where the clock of an input depends on previous values of itself?

Introduce a type which replaces $\Pi x : cl_1.cl_2$ and $\Sigma x : cl_1.cl_2$.

Causaly correct clock signatures

The type of a function f with n inputs and m output can be given the signature:

$$(x_1 : cl_1) \times ... \times (x_n : cl_n) \rightarrow (x_{n+1} : cl_{n+1}) \times ... \times (x_{n+m} : cl_{n+m})$$

where x_i (for $i \in [1..n]$) is quantified universally; (x_i) (for $i \in [n+1..n+m]$) are quantified existentially.

The signature must be syntactically causal:

- x_i can only appear in cl_j for $j > i$.
- unless it appears under a `pre` or `fby`.

that is, the clock of an input can depend on the previous value of an output.

A funny example: sorting two input streams

An example due to Ben Lippmeier (Gost motion, `https://www.gh.st`).

Consider two sorted integer input streams `left` and `right`.

Define a node `sort` which, given `left` and `right` returns a sorted stream which merge the two input streams.

```
(* Lucid Synchrone V1.1 *)
let current c default x = where
  rec o = merge c x ((default fby o) when not c)

let sort(left, right) = (c, o) where
  rec mleft = current (true fby c) left 0
  and mright = current (true fby (not c)) right 0
  and c = mleft < mright
  and o = if c then mleft else mright
```

The Lucid Synchrone V1 compiler computes:

```
val current :: (c:'a) -> 'a -> 'a on c -> 'a
val sort :: ('a on true fby c) * ('a on true fby not c)
            -> (c : 'a) * 'a
```

Properties of clock calculus

Theorem (Correctness)

Well clocked programs can be executed in a synchronous manner.

The precise formulation and proof was obtained in a very elegant manner by Boulme and Hamon [Boulme & Hamon, LPAR'01] by making a shallow embedding in Coq.

- For well clocked programs, annotate constants with their clock, e.g.,: $H \vdash 42 : b$ becomes $42[b]$ where b will be a boolean stream.
- Annotated expressions can now be given a synchronous semantics, that is, operations are applied to clocked streams.
- The clock typing rules are a direct consequence of the clocked semantics.
- If expressions are represented as Coq terms, clock rules are enforced by the typing rules of Coq.

Use of clocks

For code generation, clocks are used for control optimization. An expression with clock type s is only executed with s is true.

The explicit representation of the absent value can be removed.

Transform programs that manage streams into programs that manage streams with clocks annotations:

$$H \vdash e \ : cl \Rightarrow e'$$

expression e with clock cl is transformed into an expression e'

Annotating Expressions with their Clock

The basic language is extended with explicit annotations. pres is an enable bit. This bit is associated to every operation and register.

$$
\begin{array}{lcl}
e & ::= & i_{pres} \mid \mathtt{op}_{pres}(e, e) \mid x \\
 & & \mid \mathtt{pre}_{pres}\, e \mid e \;\mathtt{->}_{pres}\; e \\
 & & \mid \mathtt{rec}\; x.e \\
 & & \mid (e, e) \\
 & & \mid \lambda\alpha_1, ..., \alpha_n.e \\
 & & \mid \lambda x.e \mid e(e) \\
 & & \mid \mathtt{fst}\; e \mid \mathtt{snd}\; e \\
 & & \mid pres \\
 & & \\
pres & ::= & pres\; \mathtt{on}\; e \mid \alpha \mid \mathtt{true}
\end{array}
$$

Transformation

To produce a program where expressions are annotated with their clock.

$$\lambda x.(0 \;\texttt{fby}\; x) + 2 \;:\; \forall \alpha.\alpha \rightarrow \alpha$$

is translated into:

$$\lambda \alpha.\lambda x.(0_\alpha \;\texttt{fby}\; x) + 2_\alpha)$$

- An abstraction at every generalization point.
- An application at every instantiation point.
- This mechanism is necessary because several clock variables can be present in a clock scheme.
- In practice, the clock is only useful for stateful operations (`pre`, `->` and `fby`).

The Program Transformation

(Const)

$$\frac{H \vdash s \Rightarrow c_e}{H \vdash i : s \Rightarrow i[c_e]}$$

(Var)

$$\frac{cl, (c_1, ..., c_n) \in \mathit{Instanciate}(\sigma)}{H, x : \sigma \vdash x : cl \Rightarrow x\ c_1\ ...\ c_n}$$

(op)

$$\frac{H \vdash s \Rightarrow c_e \qquad H \vdash e_1 : s \Rightarrow c_1 \qquad H \vdash e_2 : s \Rightarrow c_2}{H \vdash op(e_1, e_2) : s \Rightarrow op_{c_e}(c_1, c_2)}$$

(abst)

$$\frac{H, x : cl \vdash e : cl' \Rightarrow c \qquad x \notin fv(H)}{H \vdash \lambda x.e : \forall x : cl.cl' \Rightarrow \lambda x.c}$$

(APP)

$$\frac{H \vdash f : \forall x : cl.cl' \Rightarrow f_c \qquad H \vdash e : cl \Rightarrow e_c}{H \vdash fe : cl'[e/x] \Rightarrow f_c\, e_c}$$

(REC)

$$\frac{H, x : cl \vdash e : cl \Rightarrow c \qquad x \notin fv(H)}{H \vdash \texttt{rec}\ x.e : cl \Rightarrow \texttt{rec}\ x.c}$$

Instanciation, Generalization:

$$cl[s_1/\alpha_1, ..., s_n/\alpha_n], (s_1, ..., s_n) \quad \in \quad \mathit{Instanciate}(\forall \alpha_1, ..., \alpha_n.cl)$$

$$\begin{array}{rcl} \mathit{Generalize}(H, cl) & = & \forall \alpha_1, ..., \alpha_m.cl, (\alpha_1, ..., \alpha_n) \\ & & \text{where } \{\alpha_1, ..., \alpha_n\} = FV(cl) \backslash FV(H) \end{array}$$

(LET)

$$\frac{\begin{array}{c} \sigma, (\alpha_1, ..., \alpha_n) = \mathit{Generalize}(H, cl_1) \\ H \vdash e_1 : cl_1 \Rightarrow c_1 \qquad H, x : \sigma \vdash e_2 : cl_2 \Rightarrow c_2 \end{array}}{H \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : cl_2 \Rightarrow \texttt{let } x = \lambda \alpha_1, ..., \alpha_n.c_1 \texttt{ in } c_2}$$

What is the operator On?

If s is a clock expression and e is a boolean expression, s on e is called a sub-clock of s.
s on e is true whenever e is present and true. e must be on clock s.
Thus, if s on e is true, then is s.

(ON)
$$\frac{H \vdash s \Rightarrow c_s \qquad H \vdash e : s \Rightarrow c_e}{H \vdash s \texttt{ on } e \Rightarrow c_s \texttt{ on } c_e}$$

(CLOCK-VAR)
$$H \vdash \alpha \Rightarrow \alpha$$

Algorithm and implementation choices

The very first description of this clock type system was presented at ICFP'96 [2].

- Clock type inference based on the algorithm W of ML.
- First order unification between clock, structural.
  - cl_1 `on` $e_1 \equiv cl_2$ `on` e_2 if $cl_1 \equiv cl_2$
  - e_1 and e_2 syntactically equal. The following is rejected:

    ```
    let f x =
      let z = x = 0 in
      (1 when z) + (2 when (x = 0))
    ```
- Dependences for functions ($\forall x : cl_1.cl_2$) must be in prenex form. Only the first signature is possible:

  ```
  let f x g = (g x) + (1 when x)
  f :  (x:a) -> (a -> a on x) -> a on x
  f :  (x:a) -> ((y:a) -> a on y) -> a on x
  ```

Comparison with the Lustre Clock Calculus

The system was implemented in **Lucid Synchrone** Version 1 (1996). It was kept upto Version 2 (2002).

- The ReLuC compiler of SCADE/Lustre (Esterel-Technologies) implemented a clock calculus close to the presented one.
- Clock verification instead of inference.
- A restriction in the clock type language. Clock scheme of the form $\forall\alpha.cl$ with a single clock variable.
- This is the base clock of the node.

```
let f (x,y) = (x+1, y+2)
```

*f : ('a * 'b) -> ('a * 'b)* ← *in* **Lucid Synchrone**
*f : ('a * 'a) -> ('a * 'a)* ← *in* **Lustre**

- no oversampling in **Lustre**

```
let rec half = true -> not (pre half)
let stuttering x = o where
    rec o = merge half x ((0 -> pre o) when not half)
f ::  'a on half -> 'a
```

- no polymorphic constant (they are all on the base clock of the node). The following program is rejected:

```
let rec half = true -> pre (not (half))
let f x = x when half when half
f :  'a -> 'a on half on half
```

Clock polymorphism (constants)

- Un stream defined at top level can be seen as a constant process (with no input).

  ```
  let rec half = true -> pre (not half)
  ```

  is a short-cut for (*i.e, it is compiled into*):

  ```
  let process_half () = half where
      rec half = true -> pre (not half) in half
  ```

- every instance of `half` has its own clock, thus:

  ```
  let f x = x when half when half
  ```

  is a short-cut for:

  ```
  let f x =
      (x when process_half())
              when (process_half(() when process_half()))
  ```

Conclusion

- A dependent-type system. In practice, restrict boolean expressions that appear in clock types (in s `on` e).
- The first version of the ReLuC compiler (at Esterel-Technologies) was based on this type system.
- It is possible to do a shalow embedding in Coq [Boulme & Hamon, LPAR'01]
- In 2003, we found a way to get something even simpler with a clock calculus that is amost the ML type system [Colaco and Pouzet, EMSOFT'03].
- This system was the basis of the clock calculus of Scade 6.
- This simpler system was reused and extended in two directions: the modeling and checking of periodic clocks [Julien Forget's PhD. thesis], the theory of N-synchrony [Florence Plateau's PhD. thesis, POPL'06 [3], etc.]

Sylvain Boulmé and Grégoire Hamon.

Certifying Synchrony for Free.

In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag.

Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.di.ens.fr/~pouzet/bib/bib.html`.

Paul Caspi and Marc Pouzet.

Synchronous Kahn Networks.

In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet.

N-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems.

In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

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

Gilles Kahn.
The semantics of a simple language for parallel programming.
In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

E. A. Lee and D. G. Messerschmitt.
Static scheduling of synchronous data flow programs for digital signal processing.
IEEE Trans. on Computers, 36(2), 1987.

Thomas M Park.
Bounded Scheduling of Process Networks.
PhD thesis, University of California at Berkeley, 1995.