# Clocks as First Class Abstract Types

Jean-Louis Colaço[1] and Marc Pouzet[2]

[1] ESTEREL Technologies,
Park Avenue 9, Rue Michel Labrousse 31100 Toulouse, France
Jean-Louis.Colaco@esterel-technologies.com
[2] Laboratoire LIP6, Université Pierre et Marie Curie,
8 rue du Capitaine Scott, 75015 Paris, France.
Marc.Pouzet@lip6.fr

**Abstract.** Clocks in synchronous data-flow languages are the natural way to define several time scales in reactive systems. They play a fundamental role during the specification of the system and are largely used in the compilation process to generate efficient sequential code. Based on the formulation of clocks as *dependent types*, the paper presents a simpler clock calculus reminiscent to ML type systems with first order abstract types *à la* Laufer & Odersky. Not only this system provides clock inference, it shares efficient implementations of ML type systems and appears to be expressive enough for many real applications.

## 1 Introduction

### 1.1 Synchronous Data-Flow Programming

Synchronous languages have been introduced to deal with *real-time* systems, that is, systems requiring the ability to react to their environnement in bounded time and memory. These languages introduce a discrete and logical notion of time. In this logical time, events and the reaction of the system to their occurences have to be simultaneous and instantaneous. This hypothesis of instantaneous reaction, called the *synchronous hypothesis*, allow a concurrent but deterministic description of systems and efficient compilation of programs into sequential code.

Several languages have been designed on top of the synchronous hypothesis, imperative ones (Esterel [5]), graphical ones (Argos [18], SyncCharts [1], Ptolemy [7]) or data-flow ones (Lustre [14], Signal [3], Lucid Synchrone [23,11]). Synchronous data-flow languages manage (possibly) infinite sequences or *streams* as primitive values. Synchrony appears naturally in this model: streams are time indexed sequences and at time $n$, every stream takes its $n^{th}$ value. Synchronous data-flow languages have proved to be well adapted to the design of sampled systems.

Lustre [14] has been successfully used by several industrial companies to implement safety critical systems in various domains like nuclear plants, civil aircrafts, transport and automotive systems. All these developments have been done using Scade, a graphical environment based on Lustre and distributed by Esterel Technologies [25].

R. Alur and I. Lee (Eds.): EMSOFT 2003, LNCS 2855, pp. 134–155, 2003.

### 1.2 Clocks

Synchronous systems can be described as the parallel composition of several processes [4]. Naturally, there arises a need to compose several processes computed on different rates. Rates in synchronous data-flow programming find a natural expression: the logical time defines the fastest possible rate in the system, called the *base clock*. All the other rates, or *clocks*, are derived from the base clock. The other clocks are slower than the base clock and are obtained by stating that a stream, on a given instant of the logical time, can be either absent or present. In the following example, `x` and `y` are two streams, and `y` is twice slower than `x`:

| x | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|---|
| y | 0 | | 1 | | 2 | | 3 | ... |

However, combining streams on different rates, such as `x` and `y`, can be tricky. In general, some synchronisation mechanism using possibly unbounded memory may be needed at run-time if such combinations are allowed [8]. Thus, data-flow synchronous languages provide a static analysis, checking that all combination of streams, even on different rates, can be evaluated without synchronisation mechanism (i.e, buffering mechanism). This analysis is known as the *clock calculus*.

Clocks have been originaly introduced in Lustre and Signal. In this paper, we focus our attention on the language Lucid Synchrone. Lucid Synchrone is as an extension of Lustre with features from ML-like languages, while retaining its fundamental properties: it is based on the same Kahn model [15]; it uses clocks as a way to deal with several time scales in a real-time system and programs are compiled into finite memory transition systems. Moreover, it provides powerfull extensions such as higher-order features, data-types, type and clock inference. Being an extension of Lustre, the results presented in the paper can be applied to Lustre as well. The language is used today by the Scade team at Esterel Technologies for the development of a new compiler of Scade, the industrial version of Lustre (see [12], for example).

### 1.3 Contribution

Previous works have shown that the clock calculus of Lustre could be defined as a classical *dependent type* system [10]. This clock calculus has been implemented in the first compiler of Lucid Synchrone (in 1996) and has proved to be both simple and expressive. It has served as a basis to implement a *clock verifier* inside a compiler for Scade at Esterel Technologies (in 2001). This compiler has been used in experiments on real-size examples (more than 50000 lines of code) and the dependent-based clock verifier appeared to be fast and satisfactory. Finally, a *shallow embedding* of Lucid Synchrone and its semantics into the Coq [13] proof assistant has been realised, establishing a correctness proof of the calculus [6].

Nonetheless, looking at real applications written in Scade, we observed that complex dependent clocks were not useful in many cases: most of the time,

clocks are used locally in the so-called *activation condition* as a way to sample a process and clock inference could be obtained with a simpler calculus. Thus, this paper presents a new clock calculus which is expressive enough for most applications found in SCADE. This calculus is based on classical Hindley-Milner type systems [20] with a restricted form of existential types as proposed by Laufer & Odersky [16]. This calculus shares efficient implementation techniques of ML type systems, it provides higher-order features and clock inference which is mandatory in a graphical environment such as SCADE.

Section 2 gives an overview of this new calculus. Examples will be given in LUCID SYNCHRONE syntax [1]. Examples will be kept first order such that they can be easily translated to LUSTRE syntax. In section 3, we define a (higher-order) synchronous data-flow kernel on which LUCID SYNCHRONE is based and give it a Kahn semantics and a synchronous data-flow semantics. Section 4 is devoted to the clock calculus presentation. Section 5 illustrates its expressiveness on some typical examples. In section 7, we discuss related works and we conclude in section 8.

## 2 Overview

As any synchronous data-flow language, LUCID SYNCHRONE is buit on top of a host language [2] used for writting primitive computations overs streams. Programs are written in an ML syntax but every ground type and value imported from the host language is implicitly lifted to streams. For example, `int` stands for the type of sequence of integers, `1` stands for the constant stream of values 1; `+` adds point-wise its two input streams and `if/then/else` is the point-wise conditional.

In the sequel we give on the right the graphical representation of some of the primitives in SCADE.

| | | | | |
|---|---|---|---|---|
| `1` | 1 | 1 | 1 | ... |
| `x` | $x_0$ | $x_1$ | $x_2$ | ... |
| `y` | $y_0$ | $y_1$ | $y_2$ | ... |
| `x + y` | $x_0 + y_0$ | $x_1 + y_1$ | $x_2 + y_2$ | ... |
| `c` | $t$ | $f$ | $t$ | ... |
| `if c then x else y` | $x_0$ | $y_1$ | $x_2$ | ... |

Furthermore, `pre` (for "previous") is an instant delay operator and `->` is an initialization operator. The first value of `pre y` is undefined and noted *nil* [3]. `fby` is the initialised delay [4] and satisfies $x$ `fby` $y = x$ `->` `pre` $y$.

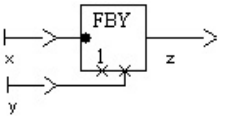
[1] A tutorial of LUCID SYNCHRONE is available at `www-spi.lip6.fr/lucid-synchrone.html`.

[2] OCAML [17] in the case of LUCID SYNCHRONE.

[3] An initialisation analysis may check that the result of a program does not depend on these *nil* values [12].

[4] The operator was first introduced in LUCID [2].

| | |
|---|---|
| `x -> y` | $x_0\ y_1\ y_2 \ldots$ |
| `pre y` | $nil\ y_0\ y_1 \ldots$ |
| `x fby y` | $x_0\ y_0\ y_1 \ldots$ |




For example, we can compute the sum (such that, $s_n = \Sigma_{i=0}^{n} x_i$) of an input sequence `x` buy writting:

```
let sum x = s where
    rec s = x -> pre s + x
node sum : int -> int
node sum :: 'a -> 'a
```

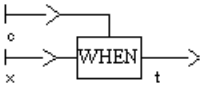
and the function which computes the minimum and the maximum of an input sequence:

```
let bounds x = (min, max) where
    rec min = x -> if x < pre min then x else pre min
    and max = x -> if x > pre max then x else pre max
node bounds : 'a -> 'a * 'a
node bounds :: 'a -> 'a * 'a
```

The two bounds are initialised to `x` and are modified step by step according to the current value of `x`. When this text is compiled, we obtain for each function a declaration (called `node` as in LUSTRE), the type (":") and clock inferences ("::"). For example, the clock `'a -> 'a` states that the function `sum` is length preserving: it returns a value every time it receives an input. Clocks are introduced below.

`when` is the *down*-sampling operator of LUSTRE which allows to extract a sub-stream from a stream according to a condition, *i.e.* a boolean stream.

| | |
|---|---|
| `base` | $t\ \ t\ \ t\ \ t \ldots$ |
| `c` | $f\ \ t\ \ f\ \ t \ldots$ |
| `x` | $x_0\ x_1\ x_2\ x_3 \ldots$ |
| `x when c` | $\quad x_1 \quad\ x_3 \ldots$ |
| `base on c` | $f\ \ t\ \ f\ \ t \ldots$ |




The sampling operator introduces *clock types*. These clock types specify time behavior of stream programs.

The clock of a sequence $s$ is a boolean sequence giving the instants where $s$ is defined. Among these clocks, the constant boolean sequence `base` denotes the base clock of the system: a sequence on the base clock is present at every instant. In the above diagram, the current value of `x when c` is present when `x` and `c` are present and `c` is true. Since `x` and `c` are on the base clock `true`, the clock of `x when c` is noted `base on c`. Now the `sum` function can be used at a slower rate by down-sampling its input stream:

```
let sc x c = sum (x when c)
node sc : int -> clock -> int
node sc :: 'a -> (_c0:'a) -> 'a on _c0
```

`sc` has a function clock which follows its type structure. This clock says that for any clock `'a`, if the first argument `x` of the function has clock `'a` and the second argument named `_c0` has clock `'a`, then the result is on clock `'a on _c0` (variables are renamed in the clock type to avoid name conflicts). An expression on clock `'a on _c0` is present when the clock `'a` is true and the boolean stream `_c0` is present and true.

Now, the sampled sum can be instanciated with an actual clock. We first define a periodic clock which is true one instant of ten.

```
(* a sampler that counts modulo n *)
let sample n = ok where
    rec cpt = 0 -> if pre cpt = n - 1 then 0
                   else pre cpt + 1
    and ok = cpt = 0
node sample : int -> bool
node sample :: 'a -> 'a

(* defining a periodic 1/10 clock *)
let clock ten = sample 10
node ten : clock
node ten :: 'a

let sum_one_over_ten x = sc x ten
node sum_one_over_ten : int -> int
node sum_one_over_ten :: 'a -> 'a on ten
```

Thus, `sum_one_over_ten x` computes the sum of the sub-stream $(x_{10n})_{n\in\mathbb{N}}$. Here, boolean streams used to sample a stream have to be first introduced with the special construction `let clock`.

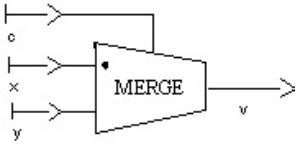
Programs must satisfy some clock constraints, as examplified on the following:

```
let unbounded x = x + (x when ten)
> x + (x when ten)
>       ^^^^^^^^^^
This expression has clock 'b on ten, but is used with clock 'b.
```

which adds the stream `x` to the sub-sequence of `x` made by filtering nine item over ten. Thus, it should compute the sequence $(x_n + x_{10n})_{n\in\mathbb{N}}$ which is clearly not bounded memory. This is why this program is rejected when dealing with real-time programming.

`merge` conversely allows slow processes to communicate with faster ones by merging sub-streams into "larger" ones:

| | |
|---|---|
| `c` | $f\ \ t\ \ f\ \ t\ \ \ldots$ |
| `x` | $\quad x_0 \quad\quad x_1\ \ldots$ |
| `y` | $y_0 \quad\quad y_1 \quad\quad \ldots$ |
| `merge c x y` | $y_0\ x_0\ y_1\ x_1\ \ldots$ |



The clock type of `merge` is:

```
let my_merge c x y = merge c x y
node my_merge : bool -> 'a -> 'a -> 'a
node my_merge :: (_c0:'a) -> 'a on _c0 -> 'a on not _c0 -> 'a
```

meaning that it combines *complementary* streams: when the second argument is present, the third one is absent and conversely.

The following operator is a classical control engineering operator, available in both the SCADE library and "digital" library of SIMULINK. This operator detects a rising edge (false to true transition). The output becomes true as soon as a transition has been detected and remains unchanged during `numberOfCycle` cycles. The output is initially false and a rising edge occuring while the output is true is detected. In LUCID SYNCHRONE syntax, this is written:

```
let count_down (reset, n) = cpt where
    rec cpt = if reset then n else (n -> pre (cpt - 1))

let risingEdgeRetrigger rer_Input numberOfCycle = rer_Output where
    rec rer_Output = (0 < v) & (c or count)
    and v = merge clk (count_down ((count,numberOfCycle) when clk))
                      ((0 fby v) whenot clk)
    and c = false fby rer_Output
    and clock clk = c or count
    and count = false -> (rer_Input & pre (not rer_Input))
```

When a clock name is introduced with the `let clock` constructor, the name is considered to be unique and does not take into account the expression on the right-hand side of the `let clock`. Thus, the following program is rejected:

```
let clock c = true in
let v = 1 when c in
let clock c = true in
let w = 1 when c in
v + w
> v + w
>     ^
This expression has clock 'a on ?_c0, but is used with clock 'a on ?_c1.
```

We stop the introductory examples here. Other examples can be found in the distribution [23]. Section 5 will give more examples of programs accepted (and rejected) with the proposed system. The main idea of this system is to replace expressions in clocks by abstract names introduced with the particular construction `let clock`. This is in contrast with previous versions of LUCID SYNCHRONE (as well as LUSTRE) where any boolean could be used to sample a stream. In term of a type system, the resulted clock calculus corresponds to a *standard* Hindley-Milner type system with a limited form of existential quantification as proposed by Laufer & Odersky [16]: the `let clock` construction corresponds to the access threw a `let` binding to a value belonging to an existential type.

## 3 A Synchronous Kernel and Its Semantics

We present here the core language used in this paper. It is a higher-order functional language over streams enriched with special operators for manipulating these streams:

$$
\begin{array}{rcl}
e & ::= & i \mid x \mid op(e,e) \mid e \ \texttt{fby}\ e \\
& & \mid e \ \texttt{when}\ e \mid e \ \texttt{whenot}\ e \mid \texttt{merge}\ e\ e\ e \\
& & \mid e(e) \mid \lambda x.e \mid \texttt{rec}\ x = e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& & \mid \texttt{let clock}\ x = e\ \texttt{in}\ e \\
& & \mid (e,e) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
i & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid ... \\
op & ::= & + \mid ...
\end{array}
$$

An expression $e$ may be an immediate constant ($i$), a variable ($x$), a point-wisely application of an operator to a tuple of inputs ($op(e,e)$) [5], an application of an initialised delay (`fby`) [6], an application of sampling operators (`when` and `whenot`) or the combination operator (`merge`), an application ($e(e)$), an abstraction $\lambda x.e$, a local definition (`let` $x = e$ `in` $e$), a local definition of a clock (`let clock` $x = e$ `in` $e$), a recursive expression (`rec` $x = e$), a pair $(e,e)$ or an application of one of the pair access functions (`fst` and `snd`).

Other classical operators can be derived from this kernel. For example:

$$
\begin{array}{lcl}
\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z & = & \texttt{let clock}\ c = x\ \texttt{in}\ \texttt{merge}\ c\ (y\ \texttt{when}\ c)\ (z\ \texttt{whenot}\ c) \\
x\ \texttt{->}\ y & = & \texttt{if true fby false then}\ x\ \texttt{else}\ y \\
\texttt{pre}\ x & = & nil\ \texttt{fby}\ x
\end{array}
$$

`->` is the initialisation operator and `pre` (for "previous") is the un-initialised delay. *nil* denotes any value with the correct type.

### 3.1 Data-Flow Kahn Semantics

We first give our core language a classical Kahn semantics [15] on sequences [7]. Let $T^\omega$ be the set of finite or infinite sequences of elements over the set $T$ ($T^\omega = T^* + T^\infty$). The empty sequence is noted $\epsilon$ and $x.s$ denotes the sequence whose head is $x$ and tail is $s$. Let $\leq$ be the prefix order over sequences, i.e., $x \leq y$ if $x$ is a prefix of $y$. The ordered set $(T^\omega, \leq)$ is a cpo. If $f$ is a continuous mapping from sequences to sequences, we shall write *fix* $f$ for the smallest fix point of $f$.

If $T_1, T_2, ...$ are sets of scalar values (typically values imported from a host language), we define the domain $V$ as the smallest set containing ${T_i}^\omega$ and closed by product and exponentiation.

For any assignment $\rho$ (mapping values to variable names) and expressions $e$, we define the denotation of an expression $e$ by $S_\rho(e)$. We first give an interpretation over sequences to every data-flow primitive and constant streams. Their definition is given in figure 1.

[5] For simplicity, we only consider binary operators in this presentation.
[6] This operator has been first introduced in Lucid [2].
[7] We keep here the original notation.

$$
\begin{array}{ll}
i^{\#} & = i.i^{\#} \\
op^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(v_1.s_1, v_2.s_2) & = (v_1\ op\ v_2).op^{\#}(s_1, s_2) \\
 & \\
\texttt{fby}^{\#}(\epsilon, ys) & = \epsilon \\
\texttt{fby}^{\#}(v.xs, ys) & = v.ys \\
 & \\
\texttt{when}^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\texttt{when}^{\#}(v_1.s_1, \mathit{true}.s_2) & = v_1.(\texttt{when}^{\#}(s_1, s_2)) \\
\texttt{when}^{\#}(v_1.s_1, \mathit{false}.s_2) & = \texttt{when}^{\#}(s_1, s_2) \\
 & \\
\texttt{merge}^{\#}(s_1, s_2, s_3) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(\mathit{true}.s_1, v_2.s_2, s_3) & = v_2.\texttt{merge}^{\#}(s_1, s_2, s_3) \\
\texttt{merge}^{\#}(\mathit{false}.s_1, s_2, v_3.s_3) & = v_3.\texttt{merge}^{\#}(s_1, s_2, s_3)
\end{array}
$$

**Fig. 1.** Data-flow Kahn semantics

- immediate values ($i$) are lifted into infinite constant streams.
- operations are applied point-wisely to their arguments.
- the initialised delay (`fby`) *conses* the head of its first input to its second input.
- `when` and `merge` are filtering and combination operators overs sequences. The operation `when` emits a sub-sequence of its inputs corresponding to the instants where the condition is true. The operation `merge` merges two sub-sequences into a sequence: it emits the current value of its second inputs when the condition is true and the current value of its third input when the condition is false.

$$
\begin{array}{rl}
S_\rho(op(e_1, e_2)) & = op^{\#}(S_\rho(e_1))(S_\rho(e_2)) \\
S_\rho(e_1\ \texttt{fby}\ e_2) & = \texttt{fby}^{\#}(S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(e_1\ \texttt{when}\ e_2) & = \texttt{when}^{\#}(S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(e_1\ \texttt{whenot}\ e_2) & = \texttt{when}^{\#}(S_\rho(e_1), S_\rho(\texttt{not}\ e_2)) \\
S_\rho(\texttt{merge}\ e_1\ e_2\ e_3) & = \texttt{merge}^{\#}(S_\rho(e_1), S_\rho(e_2), S_\rho(e_3)) \\
S_\rho(x) & = \rho(x) \\
S_\rho(i) & = i^{\#} \\
S_\rho(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = S_{\rho[x \leftarrow S_\rho(e_1)]}(e_2) \\
S_\rho(\texttt{let clock}\ x = e_1\ \texttt{in}\ e_2) & = S_{\rho[x \leftarrow S_\rho(e_1)]}(e_2) \\
S_\rho(\lambda x.e) & = \lambda y.S_{\rho[x \leftarrow y]}(e) \text{ where } y \notin Dom(\rho) \\
S_\rho(e_1(e_2)) & = S_\rho(e_1)(S_\rho(e_2)) \\
S_\rho(e_1, e_2) & = (S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(\texttt{fst}\ e) & = v_1 \text{ if } S_\rho(e) = (v_1, v_2) \\
S_\rho(\texttt{snd}\ e) & = v_2 \text{ if } S_\rho(e) = (v_1, v_2) \\
S_\rho(\texttt{rec}\ x = e) & = \rho[x \leftarrow x^\infty] \text{ where } x^\infty = \mathit{fix}\ \lambda y.S_{\rho[x \leftarrow y]}(e)
\end{array}
$$

We can easily check that the primitives are continuous for the prefix order. The denotational semantics for other constructions is defined as usual (see [24]). We recall it here shortly.

The semantics gives meaning to any Kahn network. Nonetheless, it is well known that simple data-flow networks cannot always be executed in bounded memory even when they are only composed of bounded memory operators. This problem appears when non length preserving functions (i.e, sampling functions) are considered [8] and has been exemplified in section 2.

### 3.2 Synchronous Data-Flow Semantics

In order to model synchronous execution, we describe a lower level data-flow semantics where absence of a stream is made explicit. The set of instantaneous values is enrichied with a special value *abs* representing the absence. The set of finite and infinite sequences of values belonging to $T$ complemented with the absent value is noted $T^{\omega}_{abs}$ where $T_{abs} = T \cup \{abs\}$. That is:

$$\begin{array}{ll} \mathit{Stream}\,(T) & = T.\mathit{Stream}\,(T) + \epsilon \\ \mathit{Value}\,(T) & = abs + T \\ \mathit{Clocked\text{-}Stream}\,(T) & = \mathit{Stream}\,(\,\mathit{Value}\,(T)) \\ \mathit{Clock} & = \mathit{Stream}\,(\mathit{bool}) \end{array}$$

A clocked stream (element of $\mathit{Clocked\text{-}Stream}\,(T)$) is made of present or absent values. We define the clock of a sequence $s$ as a boolean sequence (without absent values) indicating when a value is present:

$$\begin{array}{ll} \mathit{clock}\,(\epsilon) & = \epsilon \\ \mathit{clock}\,(abs.xs) & = \texttt{false}.\mathit{clock}\,(xs) \\ \mathit{clock}\,(x.xs) & = \texttt{true}.\mathit{clock}\,(xs) \end{array}$$

We give a new interpretation to constant streams, to operators applied point-wisely and to synchronous primitives. This interpretation is given in figure 2 and is discussed below:

**Constant Streams.** This operator becomes polymorphic since it may produce a value (or not) according to the environment. For this reason, we add an extra argument giving its clock. Thus, $i[s]$ denotes a constant stream with clock $s$.

**Point-Wise Application.** We must decide here what to do, in the case of a binary operator, when only one of the argument is present. Three choices are possible:

1. store the available value in a state variable implementing a FIFO queue until the other input is present;
2. generate a run-time error;
3. reject statically this situation.

$$
\begin{array}{ll}
i^{\#}[\epsilon] & = \epsilon \\
i^{\#}[true.cl] & = v.i^{\#}[cl] \\
i^{\#}[false.cl] & = abs.i^{\#}[cl] \\
 & \\
op^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(abs.s_1, abs.s_2) & = abs.op^{\#}(s_1, s_2) \\
op^{\#}(v_1.s_1, v_2.s_2) & = (v_1\, op\, v_2).op^{\#}(s_1, s_2) \\
 & \\
\texttt{fby}^{\#}(\epsilon, ys) & = \epsilon \\
\texttt{fby}^{\#}(abs.xs, abs.ys) & = abs.\texttt{fby}^{\#}(xs, ys) \\
\texttt{fby}^{\#}(x.xs, y.ys) & = x.\texttt{fby1}^{\#}(y, xs, ys) \\
\texttt{fby1}^{\#}(v, \epsilon, ys) & = \epsilon \\
\texttt{fby1}^{\#}(v, abs.xs, abs.ys) & = abs.\texttt{fby1}^{\#}(v, xs, ys) \\
\texttt{fby1}^{\#}(v, w.xs, s.ys) & = v.\texttt{fby1}^{\#}(s, xs, ys) \\
 & \\
\texttt{when}^{\#}(s_1, s_2) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
\texttt{when}^{\#}(abs.xs, abs.cs) & = abs.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, true.cs) & = x.\texttt{when}^{\#}(xs, cs) \\
\texttt{when}^{\#}(x.xs, false.cs) & = abs.\texttt{when}^{\#}(xs, cs) \\
 & \\
\texttt{merge}^{\#}(s_1, s_2, s_3) & = \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{merge}^{\#}(abs.cs, abs.xs, abs.ys) & = abs.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(true.cs, x.xs, abs.ys) & = x.\texttt{merge}^{\#}(cs, xs, ys) \\
\texttt{merge}^{\#}(false.cs, abs.xs, y.ys) & = y.\texttt{merge}^{\#}(cs, xs, ys)
\end{array}
$$

**Fig. 2.** Synchronous data-flow semantics

In the context of real-time programming, and to be coherent with LUSTRE, only the third solution is retained. The point-wise application of an operator (*op*) needs its two arguments to be on the same clock. We need a static analysis — a *clock calculus* — to insure this property. This analysis will be presented in section 4.

**Delay (`fby`).** `fby` (for "followed by") is the unitary delay: it conses the head of its first argument to its second argument. The arguments and the result of `fby` must be on the same clock. `fby` corresponds to a two-state machine: while the two arguments are absent, it emits nothing and stays in its initial state `fby`. When the two arguments become present, it emits its first argument and goes into the new state `fby1` storing the previous value of its second argument. In this state, it emits a value every time its two arguments are present.

*Sampling and Composition of Sequences (*`When` *and* `Merge`*).* The sampling operator expects two arguments on the same clock. The clock of the result depends on the boolean condition ($c$). If the argument have a clock ($cl$), the clock of the

result is $(cl$ `on` $c)$ such that:

$$\begin{array}{ll}
\texttt{on}^{\#}(true.cl, true.cs) & = true.\texttt{on}^{\#}(cl, cs) \\
\texttt{on}^{\#}(true.cl, false.cs) & = false.\texttt{on}^{\#}(cl, cs) \\
\texttt{on}^{\#}(false.cl, abs.cs) & = false.\texttt{on}^{\#}(cl, cs) \\
\texttt{on}^{\#}(cl, c) & = \epsilon \text{ if } cl = \epsilon \text{ or } c = \epsilon
\end{array}$$

We can notice that the clock is a simple sequence without absent values. The definition of `merge` states that one branch must be present when the other is absent.

Since this semantics is a partial semantics, we shall introduce clock constraints to reject programs which cannot be executed synchronously. This semantics has been described in the proof assistant Coq [6]. The use of an explicit absent value is part of the semantics of Signal [21]. Nonetheless, Signal cannot receive a Kahn semantics (functional and without absence) whereas in the case of Lustre and Lucid Synchrone, the use of absence is only done for characterising networks which can be executed synchronously.

## 4 Clock Calculus

We present now the clock calculus as a type system. The goal of the clock calculus is to produce jugments of the form $H \vdash e : cl$ meaning that "the expression $e$ has clock $cl$ in the environment $H$".

$$\begin{array}{lll}
\sigma & ::= \forall\beta_1, ..., \beta_n.\forall\alpha_1, ..., \alpha_m.\forall X_1, ..., X_k.cl & n, m, k \in I\!N \\
cl & ::= cl \to cl \mid cl \times cl \mid \beta \mid s \mid (c : s) & \\
s & ::= \texttt{base} \mid s \texttt{ on } c \mid s \texttt{ on not } c \mid \alpha & \\
c & ::= X \mid n &
\end{array}$$

$$H ::= [x_1 : \sigma_1, ..., x_n : \sigma_n]$$

Clock types are decomposed into two categories, clock schemes $(\sigma)$ and clocks $(cl)$. Clock schemes are regular clocks quantified over clock type variables $(\beta)$, stream clock type variables $(\alpha)$ and carrier variables $(X)$. Regular clocks $(cl)$ may be functional $(cl \to cl)$, products $(cl \times cl)$, variables $(\beta)$, stream clocks $(s)$ and dependences $(c : s)$. A stream clock may be the base clock (`base`), a sampled clock ($s$ `on` $c$ or $s$ `on not` $c$) or a stream clock variable $(\alpha)$. A carrier $(c)$ can be either a name $(n)$ or a carrier variable $(X)$.

We define the set of free variables $FV(cl)$ of a clock $cl$ composed of free clock variables $(\beta)$, free stream clock variables $(\alpha)$ and free carrier variables $(X)$. We extend it to clock schemes and environments. $Dom(H)$ is the domain of $H$. $N(cl)$ returns the set of abstract names from $cl$. Their definitions are classical and not given in the paper.

Programs are clocked in an initial environment $H_0$ giving clocks to synchronous primitives. To make the clock calculus shorter, a synchronous expression like $e_1$ `fby` $e_2$ is treated as a regular application of a synchronous primitive

to expressions and is clocked as if it were written $(\texttt{fby}\,(e_1))\,e_2$.

$$\begin{aligned} H_0 = [\texttt{fby} \quad & : \forall\alpha.\alpha \to \alpha \to \alpha \\ \texttt{when} \quad & : \forall\alpha.\forall X.\alpha \to (X:\alpha) \to \alpha\ \texttt{on}\ X \\ \texttt{whenot} \quad & : \forall\alpha.\forall X.\alpha \to (X:\alpha) \to \alpha\ \texttt{on not}\ X \\ \texttt{merge} \quad & : \forall\alpha.\forall X.(X:\alpha) \to \alpha\ \texttt{on}\ X \to \alpha\ \texttt{on not}\ X \to \alpha \\ \texttt{fst} \quad & : \forall\beta_1,\beta_2.\beta_1 \times \beta_2 \to \beta_1, \\ \texttt{snd} \quad & : \forall\beta_1,\beta_2.\beta_1 \times \beta_2 \to \beta_2] \end{aligned}$$

The first entry in this initial environment states that the clock of $x$ `fby` $y$ is the one of $x$ and $y$. A boolean expression $e$ with clock $(c:s)$ states that $e$ is present when $s$ is true and has the abstract value $c$. Thus, an expression $e_1$ `when` $e_2$ is well clocked if the two inputs are synchronous on the clock $\alpha$. In that case, the clock of the result is a subclock $\alpha$ `on` $X$ of $\alpha$ where $X$ stands for the value of $e_2$. An expression `merge` $e\ e_1\ e_2$ is well clocked and on clock $\alpha$ if $e$ is on clock $\alpha$ and is equal to some $X$, $e_1$ is on clock $\alpha$ `on` $X$ and $e_2$ on clock $\alpha$ `on not` $X$.

Clocks may be instanciated and generalised in the following way:

$$inst(\forall \boldsymbol{\beta}.\boldsymbol{\alpha}.\boldsymbol{X}.cl') = cl'[\boldsymbol{cl}/\boldsymbol{\beta}][\boldsymbol{s}/\boldsymbol{\alpha}][\boldsymbol{c}/\boldsymbol{X}]$$

$$\begin{aligned} gen_H(cl) \quad & = \forall\beta_1,...,\beta_n.\forall\alpha_1,...,\alpha_m.\forall X_1,...,X_k.cl \\ & \quad \text{where } \beta_1,...,\beta_n,\alpha_1,...,\alpha_m,X_1,...,X_k = FV(cl)\backslash FV(H) \end{aligned}$$

It states that a clock scheme can be instanciated by replacing clock variables by clocks, stream clock variables by stream clocks and carrier variables by carriers. Moreover, any variable can be generalized as soon as it does not appear free in the environment. The clocking rules are now given in figure 3.

- a constant stream may have any clock (rule (IM))
- the inputs of imported primitives must all be on the same clock (rule (OP))
- the rules for variables (INST), functions (FUN), applications (APP), local definitions (LET) and products (PAIR) are the classical typing rules of ML.
- a clock definition `clock` $x = e$ defines a new clock name $n$ which has the clock type $s$. In doing this, $n$ must be a fresh name, that is, it must not appear free in the environment nor in the returned clock $cl_2$. It states that if $x$ evaluates to some value $n$ and is on clock $s$, then the clock of $e_2$ is $cl_2$. The symbol $n$ is an abstraction of the actual value of $e_1$ and is considered to be unique.

The clocking rule for the `let clock` construction is a particular case of the typing rule for `let` in the Laufer & Odersky type system. Here, the *abstract name* $n$ corresponds to the introduction of a fresh skolem name.

### 4.1 Correctness Theorem and the Use of Clocks

The clock calculus is used for giving a synchronous data-flow semantics to programs and to establish a correctness property: well clocked programs can be executed synchronously in the synchrounous data-flow semantics without raising

$$\text{(IM)}\ H \vdash i : s \qquad \text{(OP)}\ \frac{H \vdash e_1 : s \quad H \vdash e_2 : s}{H \vdash op(e_1, e_2) : s} \qquad \text{(APP)}\ \frac{H \vdash e_1 : cl_2 \to cl_1 \quad H \vdash e_2 : cl_2}{H \vdash e_1\,(e_2) : cl_1}$$

$$\text{(FUN)}\ \frac{H, x : cl_1 \vdash e : cl_2}{H \vdash \lambda x.e : cl_1 \to cl_2} \qquad \text{(INST)}\ \frac{cl = inst(H(x))}{H \vdash x : cl} \qquad \text{(REC)}\ \frac{H, x : s \vdash e : s}{H \vdash \texttt{rec}\ x = e : s}$$

$$\text{(LET)}\ \frac{H \vdash e_1 : cl_1 \quad H, x : gen_H(cl_1) \vdash e_2 : cl_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : cl_2} \qquad \text{(PAIR)}\ \frac{H \vdash e_1 : cl_1 \quad H \vdash e_2 : cl_2}{H \vdash (e_1, e_2) : cl_1 \times cl_2}$$

$$\text{(LET-clock)}\ \frac{H \vdash e_1 : s \quad H, x : (n : s) \vdash e_2 : cl_2 \quad n \notin N(H, cl_2)}{H \vdash \texttt{let clock}\ x = e_1\ \texttt{in}\ e_2 : cl_2}$$

**Fig. 3.** The Clock Calculus

execution errors. In practice, the clock information is used in synchronous compilers in the optimization process in order to avoid the expensive representation of "presence/absense" at run-time. Once the clock analysis is performed, every computation is annotated with its clock which serves as a guard: a computation is made only when its clock is true.

A detailed presentation of the use of clocks in the compilation process is out of scope of this paper. We simply define a translation of programs into programs where constants are annotated with their clocks. This translation follows exactly the clock calculus and is obtained by asserting the judgment:

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

meaning that the expression $e$ with clock $cl$ can be transformed into the expression $ce$. The goal of this transformation is to produce a new program where expressions are annotated with their clocks. For example:

$$f = \lambda x.(0\ \texttt{fby}\ x) + 2 : \forall\alpha.\alpha \to \alpha$$

where $+$ is the point-wise sum will be transformed into:

$$f = \lambda\alpha.\lambda x.(0[\alpha]\ \texttt{fby}\ x) + 2[\alpha]$$

The function $f$ whose clock is polymorphic receives an extra argument $\alpha$ giving its clock. Then at instanciation point, the function will receive its clock as an extra argument. For example, if $x$ is on clock $s_1$:

$$f\,(x\ \texttt{when}\ c)$$

is transformed into:

$$f\,s_1\,(x\ \texttt{when}\ c)$$

Our translation corresponds to the classical "library" method for compiling type classes in Haskell and first introduced in [26]. Clock variables are then abstracted at generalisation points and instanciated at application points. We first

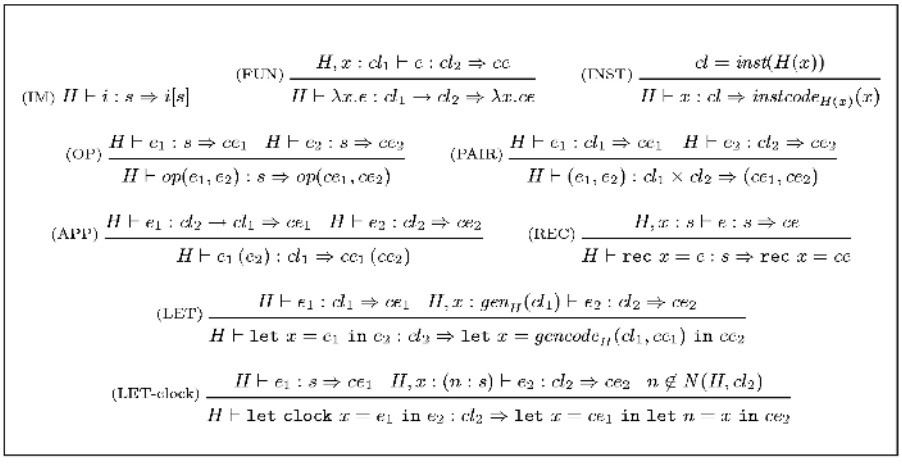



**Fig. 4.** Transforming streams into clocked-streams

introduce two auxiliary functions:

$$
\begin{array}{ll}
gencode_H(cl, ce) & = \lambda\alpha_1, ..., \alpha_m.X_1, ..., X_k.ce \\
 & \quad \text{where } \beta_1, ..., \beta_n, \alpha_1, ..., \alpha_m, X_1, ..., X_k = FV(cl) \backslash FV(H) \\
instcode_{cl}(x) & = x \\
instcode_{\forall \boldsymbol{\beta}.\boldsymbol{\alpha}.\boldsymbol{X}.cl'}(x) & = x\ s_1 ... s_n.c_1 ... c_k \text{where } inst(\forall \boldsymbol{\beta}.\boldsymbol{\alpha}.\boldsymbol{X}.cl') = cl'[\boldsymbol{cl}/\boldsymbol{\beta}][\boldsymbol{s}/\boldsymbol{\alpha}][\boldsymbol{c}/\boldsymbol{X}]
\end{array}
$$

The predicate is defined in figure 4.

- immediate constants receive an extra argument giving their clock.
- rules for abstractions (FUN), applications (APP), recursions (REC), products (PAIR) are simple morphisms.
- clocks are passed at instantiation points (rule (INST)) and abstracted at generalization points (rule (LET)). Here, only stream clock variables ($s$) or carrier variables ($X$) from the scheme clock are used since only they may appear in the sampling of a stream value.
- for clock definitions (rule (LET-clock)), the name $n$ is introduced in the generated code since it may appear in some computation of a clock.

We then prove that this transformation will not provide programs producing incomplete pattern failure. For this purpose, we define a valuation $\mathcal{V}$ from variables to boolean streams and lift it to clocks such that:

$$
\begin{array}{ll}
\mathcal{V}(\mathtt{base}) & = \mathtt{true} \\
\mathcal{V}(s\ \mathtt{on}\ c) & = \mathtt{on}^{\#}(\mathcal{V}(s), \mathcal{V}(c)) \\
\mathcal{V}(s\ \mathtt{on\ not}\ c) & = \mathtt{on}^{\#}(\mathcal{V}(s), \mathtt{not}^{\#}(\mathcal{V}(c)))
\end{array}
$$

We define interpretation functions $\mathcal{I}_{.}(.)$ relating clock schemes and set of values. An interpretation is such that:

$$
\begin{array}{ll}
v \in \mathcal{I}_{\mathcal{V}}(s) & \text{iff } \mathit{clock}\,(v) \leq \mathcal{V}(s) \text{ where } \leq \text{ stands for the prefix order} \\
v \in \mathcal{I}_{\mathcal{V}}((c:s)) & \text{iff } v \in \mathcal{I}_{\mathcal{V}}(s) \text{ and } v = \mathcal{V}(c) \\
v \in \mathcal{I}_{\mathcal{V}}(cl_1 \to cl_2) & \text{iff for all } v_1 \text{ such that } v_1 \in \mathcal{I}_{\mathcal{V}}(cl_1), v(v_1) \in \mathcal{I}_{\mathcal{V}}(cl_2) \\
(v_1, v_2) \in \mathcal{I}_{\mathcal{V}}(cl_1 \times cl_2) & \text{iff } v_1 \in \mathcal{I}_{\mathcal{V}}(cl_1) \text{ and } v_2 \in \mathcal{I}_{\mathcal{V}}(cl_2) \\
v \in \mathcal{I}_{\mathcal{V}}(\forall \beta_1, ..., \beta_k.\sigma) & \text{iff for all } cl_1, ..., cl_k, v \in \mathcal{I}_{\mathcal{V}}(\sigma[cl_1/\beta_1, ..., cl_k/\beta_k]) \\
v \in \mathcal{I}_{\mathcal{V}}(\forall \alpha_1, ..., \alpha_n.X_1, ..., X_k.\sigma) & \text{iff for all } s_1, ..., s_n, c_1, ..., c_k, \\
 & \quad v\,(\mathcal{V}s_1), ..., (\mathcal{V}s_n)\,(\mathcal{V}c_1), ..., (\mathcal{V}c_k) \in \\
 & \quad \mathcal{I}_{\mathcal{V}}(cl\sharp_1/\alpha_1, ..., s_n/\alpha_n, c_1/X_1, ..., c_k/X_k])
\end{array}
$$

**Theorem 1 (Clock Soundness).** *If* $[x_1 : \sigma_1, ..., x_n : \sigma_n] \vdash e : cl \Rightarrow ce$ *then for all valuation* $\mathcal{V}.$*, for all interpretation function* $\mathcal{I}_{.}(.)$*, for all* $v_1, ..., v_n$ *such that* $v_1 \in \mathcal{I}_{\mathcal{V}}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}}(\sigma_n)$*, we have* $\mathrm{S}_{\mathcal{V}[v_1/x_1,...,v_n/x_n]}(ce) \in \mathcal{I}_{\mathcal{V}}(cl)$.

The proof is given in appendix A.

The clock calculus presented in this paper is implemented in the Lucid Synchrone compiler and is in used for more than one year. Classical implementation techniques for ML type systems have been used (e.g, destructive unification). The technique for checking that the introduced name in the rule (LET-clock) is a fresh name and does not escape its scope is due to Pottier [19] and is used for the efficient implementation of the Laufer & Odersky system.

Practical experiments show that the clock calculus is as fast as the type inference which means that it can be applied to real-size examples.

## 5 Examples

We illustrate the expressivity of this calculus on some typical examples. A Lucid Synchrone program is a sequence of global declarations. A global declaration defines a name which can be used after its declaration. Every global declaration is analysed sequentially and compiled.

### 5.1 Activation Conditions

A classical primitive in a block diagram framework (such as the one of Scade) is the *activation condition* (also known as *enable sub-system* in Simulink). It consists of executing a node only when a condition is true. In term of clocks, it corresponds to filtering the input of that node on a certain clock `clk` and to project the result on the base clock. Such an operation is a higher-order construct and can be defined like the following:

```
let condact clk f input init =
    let rec u = merge clk (f (input when clk))
                          ((init fby u) whenot clk) in
    u
node condact : clock -> ('a -> 'b) -> 'a -> 'b -> 'b
node condact :: (clk:'a) -> ('a on clk -> 'a on clk) -> 'a -> 'a -> 'a
```

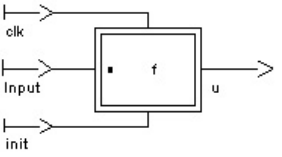



**Fig. 5.** SCADE notation for activation condition.

Its graphical representation in SCADE is given in figure 5. Using the `condact` primitive, we can rewrite the *rising edge retrigger* as it is written in the SCADE library.

```
let risingEdgeRetrigger rer_Input numberOfCycle = rer_Output where
    rec rer_Output = (0 < v) & (c or count)
    and v = condact clk count_down (count,numberOfCycle) 0
    and c = false fby rer_Output
    and clock clk = c or count
    and count = false -> (rer_Input & pre (not rer_Input))
```

Its graphical representation is given in figure 6.

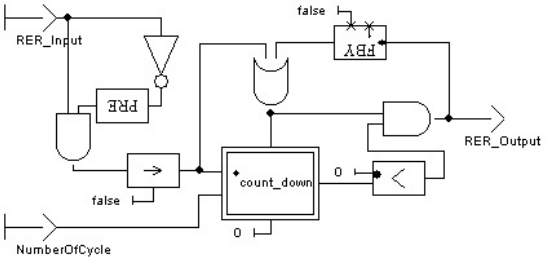



**Fig. 6.** The `risingEdgeRetrigger` in SCADE

### 5.2 Iteration

The symetric operation of the activation condition is an *iterator*. It corresponds to the writting of an internal `for` or `while` loop. This operator consists in iterating a function on an input.

```
let iter clk init f input =
    let rec o = f i
    and i = merge clk input ((init fby o) whenot clk) in
    o when clk
node iter :: (clk:'a) -> 'a -> ('a -> 'a) -> 'a on clk -> 'a on clk
```

### 5.3 Scope Restriction

Being based on the Laufer & Odersky type system, the clock calculus suffers from the same limitations. Mainly, when clock names are introduced with the `let clock` construction, these names must not escape their lexical scope. For example:

```
let escape x =
    let clock c = (x = 0) in
    x when c

>....escape x =
> let clock c = (x = 0) in
> x when c
The clock of this expression depends on ?c_0 which escape its scope.
```

Thus, the clock calculus is less expressive than the previous clock calculus of Lucid Synchrone or the one of Lustre where a result can depend on a clock computed internally as soon as the clock is returned as a result. This could be considered as a *serious* restriction. Quite surprisingly, this is not the case in practice mostly because most (near all!) programs found in Scade use clocks in a limited way corresponding to the *activation condition* and because these languages do not provide higher-order features.

## 6 Extension: Clocks Defined at Top-Level

We have defined (and implemented) an extension of the presented clock calculus with *top-level clocks*. We call *top-level* or *constant* clocks, clocks defined globally at top-level of a program. These clocks do not depend on any input of the program but they may, themselves, be instanciated on different clocks. Consider, for example:

```
let clock sixty = sample 60 (* 1/60 *)
node sixty : clock
node sixty :: 'a
```

It defines a periodic clock `sixty`. Using it, we can define a (real) clock in the following way:

```
let hour_minute_second second =
    let minute = second when sixty in
    let hour = minute when sixty in
    hour,minute,second

node hour_minute_second : 'a -> 'a * 'a * 'a
node hour_minute_second ::
     'a -> 'a on sixty on sixty * 'a on sixty * 'a
```

A stream on clock `'a on sixty on sixty` is only present one instant over 3600 instants which match perfectly what we are expecting.

Using clocks defined globally, we can write simple over-sampling functions. Consider, for example, the computation of the sequence $(o_n)_{n\in I\!N}$ such that:

$$o_{2n} = x_n$$
$$o_{2n+1} = x_n$$

It can be programmed in the following way:

```
let clock half = h where
    rec h = true -> not (pre h)
let stuttering x = o where
    rec o = merge half x (0 -> (pre o) whenot half)
node stuttering :: 'a on half -> 'a
```

This is a true example of oversampling, that is, a function whose internal clock is *faster* than the clock of its input. This example shows that some limited form of oversampling — which is possible in SIGNAL and not in LUSTRE — can be achieved with simple typing techniques.

It appears that top-level clocks are sufficient to express many programs appearing in the SCADE environment. In particular, many clocks are periodic [8]. Periodic (constant) clocks are useful for specifying hard real-time constraints from the environment and to direct the scheduling strategy of the compiler [9]. It is an open question to know whether constant clocks — which only need a very modest, dependent-less type system — are sufficient for this purpose.

In term of type system, these constant clocks defined at top-level do not raise any technical difficulty. Clock names defined at top-level are constant names and act as new type constructors with arity zero defined in ML languages. Then, as it is the case for clock names defined locally, two clocks are equal if they have the same name.

## 7 Related Works

LUCID SYNCHRONE has been originally developed as a functional extension of LUSTRE and to serve as an experimental language for prototyping extensions for it. It is based on the same semantic model and clocks are largely reminiscent of the ones in LUSTRE. Up to syntactic details, the clock calculus presented in this paper can be applied directly to LUSTRE programs. Nonetheless, the clock calculus is expressed here as a typing problem: this allows clocks to be automatically inferred using standard techniques and is compatible with higher-order. In comparison, LUSTRE is first-order and clocks are verified instead of being inferred. Clock inference is mandatory in a graphical programming environment such as the one of SCADE (the clock of intermediate wires must be computed

[8] Typically, an application is made of several main behaviors, each of them being executed at different periodic (constant) clocks, e.g., corresponding to 60hz, 200hz.

automatically) and one of our motivation is to design an efficient clock inference mechanism for SCADE. Finally, higher-order features appeared to be very useful for the prototyping of special purpose primitives before their ad-hoc incoding inside the SCADE compiler.

The present clock calculus can also be related to the one of SIGNAL. Clocks in SIGNAL can be arbitrary boolean expressions and the language supports full over-sampling. As a result, the clock calculus of SIGNAL reaches an impressive expressive power. Two streams with unrelated clocks can be combined thru the operator `default`, and the clock of the resulting stream is the "union" of the clocks of the two arguments. However, this expressiveness comes at the price of a greater complexity. In particular, the clock calculus of SIGNAL calls for boolean resolution techniques and fix-point iteration whereas we use simpler unification techniques.

This new calculus is less expressive than the previous clock calculus of LUCID SYNCHRONE, based on dependent types. The main difference is that clock types only contain *abstract names* introduced with the special primitive `let clock` whereas any boolean expression could be used for sampling a stream. From a SCADE user point of view, the need to name each clock and introduce it with a special construct (here `let clock`) is not a problem; people developing safety critical applications are pretty familiar with this kind of discipline.

## 8 Conclusion

In this paper we have presented a simple type-based clock inference calculus for a synchronous data-flow language providing higher-order features such as LUCID SYNCHRONE. The system is based on the extension of an ML type system with first class abstract types proposed by Laufer & Odersky. This calculus has been obtained by forbiding general boolean expressions in clock types, allowing them to contain only abstract names. These abstract names denote special boolean streams which are used to sample a stream. They can be introduced through the dedicated construction `let clock`.

We discovered recently that the idea of replacing boolean expressions by names in clocks has been already suggested by other implementors of synchronous compilers [22]. Nonetheless, it does not seem that the resulting clock calculus has been identified nor implemented.

Our motivation in doing such a clock calculus was mainly pragmatic. After several years of use of a dependent-type based clock calculus and looking at programs written in SCADE and LUSTRE, we observed that most of the time complex dependences in clock are useless and that the simplification proposed here is expressive enough for many real applications. The new system is simpler to use and it shares standard theory and implementation techniques of ML type systems.

Finally, we believe that bridging together the clock calculus and standard ML typing may contribute to the use of clocks as a good programming discipline in synchronous data-flow languages.

## References


1. Charles André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *CESA*, Lille, july 1996. IEEE-SMC. Available at: `www-mips.unice.fr/~andre/syncharts.html`.
2. E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language*. Academic Press, 1985.
3. A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
4. G. Berry. Real time programming: Special purpose or general purpose languages. *Information Processing*, 89:11–17, 1989.
5. G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
6. 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 `http://www-spi.lip6.fr/lucid-synchrone`.
7. J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of computer Simulation*, 1994. special issue on Simulation Software Development.
8. P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.
9. P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, P. Niebert From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. LCTES'03.
10. Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.
11. Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. submitted to publication, 2001.
12. Jean-Louis Colaço and Marc Pouzet. Type-based Initialization of a Synchronous Data-flow Language. In *Synchronous Languages, Applications, and Programming*, volume 65. Electronic Notes in Theoretical Computer Science, 2002.
13. The coq proof assistant, 2002. `http://coq.inria.fr`.
14. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
15. G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.
16. Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.
17. Xavier Leroy. The Objective Caml system release 3.06. Documentation and user's manual. Technical report, INRIA, 2002.
18. Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.

19. Michel Mauny and François Pottier. An implementation of Caml Light with existential types. Technical Report 2183, INRIA, October 1993.
20. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.
21. D. Nowak, J. R. Beauvais, and J. P. Talpin. Co-inductive axiomatisation of synchronous language. In *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'98)*. Springer Verlag, October 1998.
22. Lecheck Olendersky. Private communication, December 2002. Workshop Synchron, Toulon, France.
23. Marc Pouzet. *Lucid Synchrone, version 2. Tutorial and reference manual.* Université Pierre et Marie Curie, LIP6, Mai 2001. Distribution available at: `www-spi.lip6.fr/lucid-synchrone`.
24. John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.
25. SCADE. `http://www.esterel-technologies.com/scade/`.
26. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

## A Proof of Theorem 1 (Clock Soundness)

The property is proved by induction on the structure of expressions.

*Case* $e' = i$. We have $\vdash i : s \Rightarrow i[s]$. By definition, $clock\,(S_{\mathcal{V}}(i[s])) = \mathcal{V}(s)$.

*Case* $e' = x$. We have $H, x : \sigma \vdash x : cl \Rightarrow instcode_H(x)$. There are several cases. Either $\sigma = cl$ or some variables are universally quantified.

- If $\sigma = cl$ then $H, x : cl \vdash x : cl \Rightarrow x$ and the property holds by definition.
- Let $\sigma = \forall\beta_1, ..., \beta_n.\alpha_1, ..., \alpha_m.X_1, ..., X_k.cl$. Let $H = [\sigma_1/x_1, ..., \sigma_p/x_p, \sigma/x]$ be the typing environment and $[v_1/x_1, ..., v_p/x_p, v/x]$ the corresponding environment for the evaluation. Let $\rho = \mathcal{V}[v_1/x_1, ..., v_p/x_p, v/x]$.

  Let $v$ such that $v \in \mathcal{I}_{\mathcal{V}}(\forall\beta_1, ..., \beta_n.\alpha_1, ..., \alpha_m.X_1, ..., X_k.cl)$. According to the definition of $\mathcal{I}_{\mathcal{V}}(.)$, for all $cl_1, ..., cl_n, s_1, ..., s_m, c_1, ..., c_k$, we have $v\,(\mathcal{V}s_1)...(\mathcal{V}s_n).(\mathcal{V}c_1)...(\mathcal{V}c_k) \in \mathcal{I}_{\mathcal{V}}(cl[cl_1/\beta_1, ..., cl_n/\beta_n][s_1/\alpha_1, ..., s_m/\alpha_m][c_1/X_1, ..., c_k/X_k])$.
  The property holds since:
  $v\,(\mathcal{V}s_1)...(\mathcal{V}s_n).(\mathcal{V}c_1)...(\mathcal{V}c_k) = (S_\rho(x\,s_1...s_n.c_1...c_k)) = S_\rho(instcode_H(x))$.

*Case of Primitives.* Direct recurrence.

*Case* $e' = e_1(e_2)$. Let $H$ be the typing environment and $\rho$, the corresponding evaluation environment. Suppose the property holds for $e_1$ and $e_2$, that is $S_{\mathcal{V}}(ce_1) \in \mathcal{I}_{\mathcal{V}}(cl_1 \to cl_2)$ and $S_V(ce_2) \in \mathcal{I}_{\mathcal{V}}(cl_1)$. According to the definition of the interpretation $\mathcal{I}_{\mathcal{V}}(.)$, for any $v \in \mathcal{I}_{\mathcal{V}}(cl_1)$ we have $(S_{\mathcal{V}}(ce_1))(v) \in \mathcal{I}_{\mathcal{V}}(cl_2)$. Thus, $(S_{\mathcal{V}}(ce_1))(S_{\mathcal{V}}(ce_2)) = S_{\mathcal{V}}(ce_1(ce_2)) \in \mathcal{I}_{\mathcal{V}}(cl_2)$. Thus the property holds.

*Case* $e' = \lambda y.e$. Let $H$ be the typing environment and $\rho$, the corresponding evaluation environment. Suppose that $H, y : cl_y \vdash e : cl \Rightarrow ce$. Applying the recurrence hypothesis, for all $\rho$ corresponding to $H$ and for all $v_y \in \mathcal{I}_{\mathcal{V}}(cl_y)$ we have $S_{\mathcal{V}[v_y/y]}(ce) \in \mathcal{I}_{\mathcal{V}}(cl)$. Thus, for all $v_y \in \mathcal{I}_{\mathcal{V}}(cl_y)$, we have $S_{\mathcal{V}[v_y/y]}((\lambda y.ce)\, y) \in \mathcal{I}_{\mathcal{V}}(cl)$. Thus, for all $v_y \in \mathcal{I}_{\mathcal{V}}(cl_y)$, we have $(S_{\mathcal{V}}(\lambda y.ce))(v_y) \in \mathcal{I}_{\mathcal{V}}(cl)$. Thus the property holds.

*Case of* $e' = \texttt{let}\ y = e_1\ \texttt{in}\ e_2$. Direct recurrence and combination of the preceding rules.

*Case of Clock Declarations* $e = \texttt{let clock}\ x = e_1\ \texttt{in}\ e_2$. Suppose that the property holds for $H \vdash e_1 : s_1 \Rightarrow ce_1$, that is, for all $\mathcal{V}$ and $[v_1/x_1, ..., v_n/x_n]$ such that $v_1 \in \mathcal{I}_{\mathcal{V}}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}}(\sigma_n)$, $S_{\mathcal{V}[v_1/x_1,...,v_n/x_n]}(ce_1) \in \mathcal{I}_{\mathcal{V}}(s_1)$. Let $n$ be a fresh name $n \notin N(H)$. Let $\mathcal{V}'$ be an extension of $\mathcal{V}$ such that $\mathcal{V}'(z) = \mathcal{V}(z)$ if $z \neq n$. Then for all $\mathcal{V}'$ extending $\mathcal{V}$, for all $v_1 \in \mathcal{I}_{\mathcal{V}'}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}'}(\sigma_n)$ we have $S_{\mathcal{V}'[v_1/x_1,...,v_n/x_n]}(ce_1) \in \mathcal{I}_{\mathcal{V}'}(s_1)$.

Suppose that the property holds for $H, x : (n : s_1) \vdash e_2 : cl_2 \Rightarrow ce_2$ where $n \notin N(H)$ and $n \notin N(cl_2)$, that is for all $\mathcal{V}'$, for all $v_1 \in \mathcal{I}_{\mathcal{V}'}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}'}(\sigma_n), v_x \in \mathcal{I}_{\mathcal{V}'}((n : s_1))$, we have $S_{\mathcal{V}'[v_1/x_1,...,v_n/x_n,v_x/x]}(ce_2) \in \mathcal{I}_{\mathcal{V}'}(cl_2)$. $v_x \in \mathcal{I}_{\mathcal{V}'}((n : s_1))$ means that $v_x \in \mathcal{I}_{\mathcal{V}'}(s_1)$ and $\mathcal{V}'(n) = v_x$. Since $n$ is a fresh name, this means that for all $z$ $\mathcal{V}'(z) = \mathcal{V}(z)$ is $z \neq n$ and $\mathcal{V}'n = v_x$ otherwise, that $S_{\mathcal{V}'[v_1/x_1,...,v_n/x_n,v_x/x]}(ce_2) = S_{\mathcal{V}[v_1/x_1,...,v_n/x_n,v_x/x,v_x/n]}(ce_2)$ and finally, that $\mathcal{I}_{\mathcal{V}'}(cl_2) = \mathcal{I}_{\mathcal{V}}(cl_2)$. Thus, $S_{\mathcal{V}[v_1/x_1,...,v_n/x_n,v_x/x,v_x/n]}(ce_2) \in \mathcal{I}_{\mathcal{V}}(cl_2)$, that is $S_{\mathcal{V}[v_1/x_1,...,v_n/x_n]}(\texttt{let}\ x = ce_1\ \texttt{in let}\ n = x\ \texttt{in}\ ce_2) \in \mathcal{I}_{\mathcal{V}}(cl_2)$ which is the expected result.

*Case of Recursions* $e' = (\texttt{rec}\ x = e)$. Let $f : y \mapsto S_{\mathcal{V}[y/x]}(ce)$. We have $\epsilon \in \mathcal{I}_{\mathcal{V}}(s)$. For all $v_1 \in \mathcal{I}_{\mathcal{V}}(\sigma_1), ..., v_n \in \mathcal{I}_{\mathcal{V}}(\sigma_n), v \in \mathcal{I}_{\mathcal{V}}(s)$, $S_{\mathcal{V}[v/x]}(ce) \in \mathcal{I}_{\mathcal{V}}(s)$, that is $f(v) \in \mathcal{I}_{\mathcal{V}}(s)$. Thus, for all $n$, $f^n(\epsilon) \in \mathcal{I}_{\mathcal{V}}(s)$. For all $n$, $f^n(\epsilon) \leq lim_{n\to\infty}(f^n(\epsilon))$. $\lim_{n\to\infty}(f^n(\epsilon)) \notin \mathcal{I}_{\mathcal{V}}(s)$, means that *clock* $(\lim_{n\to\infty}(f^n(\epsilon))) \not\leq \mathcal{V}(s)$. Thus, there exists a $k$ such that $f^k(\epsilon) \not\leq \mathcal{V}(s)$ which is contradictory. Thus, we get the expected result. □