Lucid Synchrone
a Functional Synchronous Language

Marc Pouzet
LIENS
`Marc.Pouzet@ens.fr`

Modelica Design Meeting
Munich, January 25th, 2011.

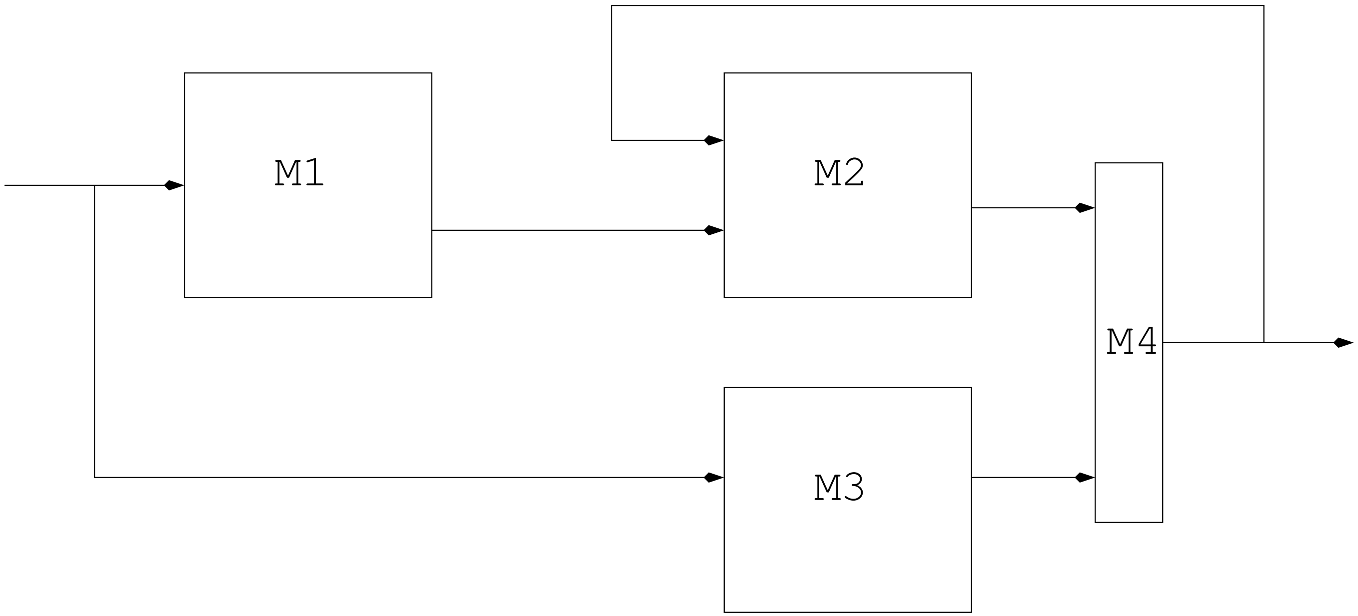
Reactive systems

- They react continuously to the external environment.
- At the speed **imposed** by this environment.
- **Statically bounded** memory and response time.

Conciliate three notions in the programming model:

- Parallelism, concurrency while preserving determinism.
  e.g, control at the same time rolling and pitching
  ↪ **parallel description of the system**
- Strong temporal constraints.
  e.g, the physics does not wait!
  ↪ **temporal constraints should be expressed in the system**
- Safety is important (critical systems).
  ↪ **well founded languages, verification methods**

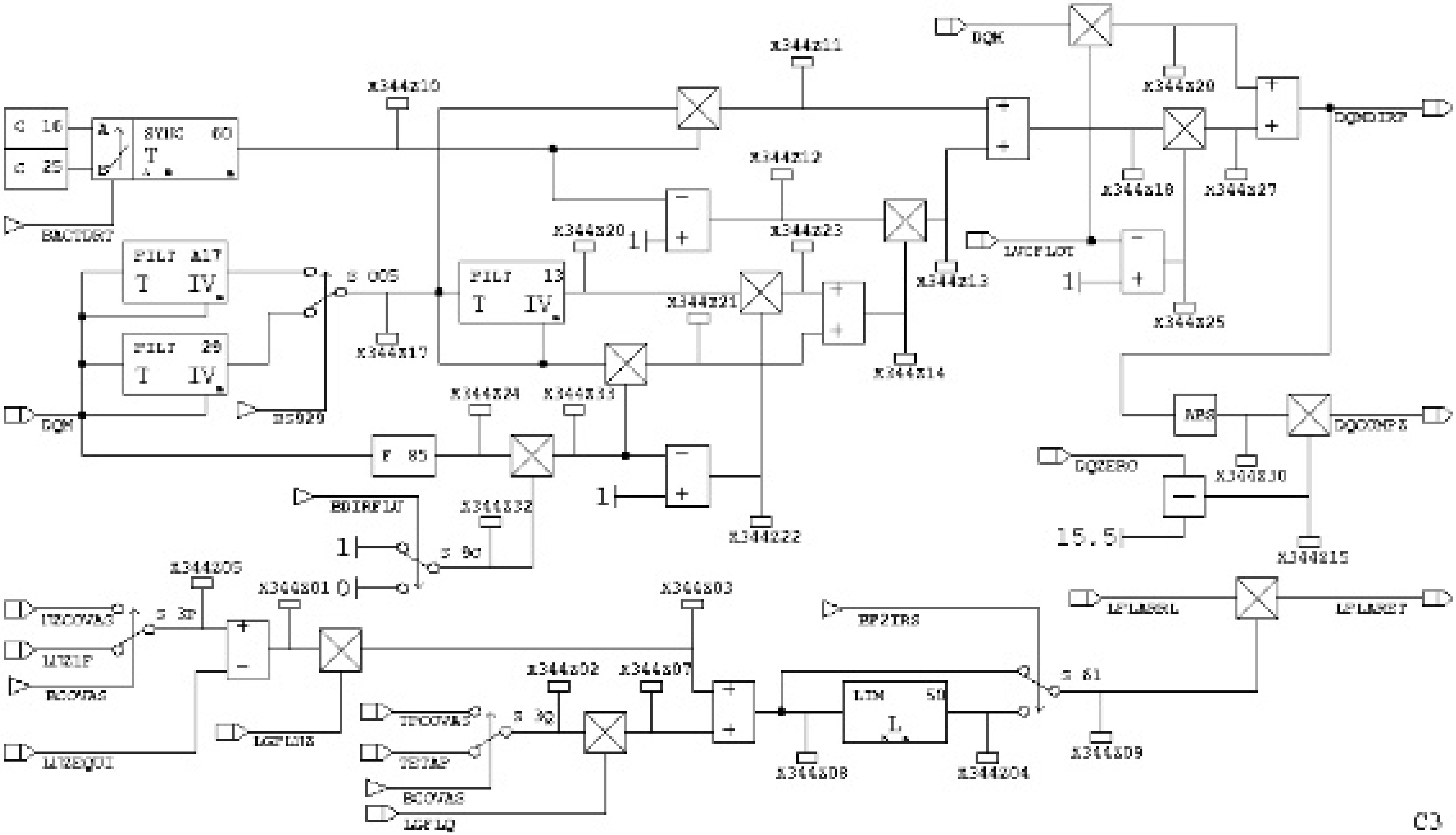
Synchronous Kahn Networks



- **parallel processes** communicating through data-flows
- **communication in zero time**: data is available as soon as it is produced.
- a **global logical time scale** even though individual rhythms may differ
- these drawings are not so different from actual computer programs

SAO (Spécification Assistée par Ordinateur) — Airbus 80's

Describe the system as block diagrams (synchronous communicating machines)



Programming with data-flow equations

The language Lustre (Caspi & Halbwachs, 1984).

X	1	2	1	4	5	6	...
Y	2	4	2	1	1	2	...
1	1	1	1	1	1	1	...
$X + Y$	3	6	3	5	6	8	...
$X + 1$	2	3	2	5	6	7	...

The equation $Z = X + Y$ means that at every instant n, $Z_n = X_n + Y_n$.

Time is logical: the two inputs X and Y arrive "at the same time"; the output Z is produced at the very same instant.

Practically speaking, it suffices to check that the current output is produced before the input for the next instant arrives.

Memorizing values

We add operators to memorize the value produced at the previous instant.

`X`	1	2	1	4	5	6	...
`pre X`	*nil*	1	2	1	4	5	...
`Y`	2	4	2	1	1	2	...
`Y -> pre X`	2	1	2	1	4	5	...
`S`	1	3	4	8	13	19	...

The sequence (S_n) such that $S_0 = X_0$ and $S_n = S_{n-1} + X_n$ for all $n > 0$ is written:

```
S = X -> pre S + X
```

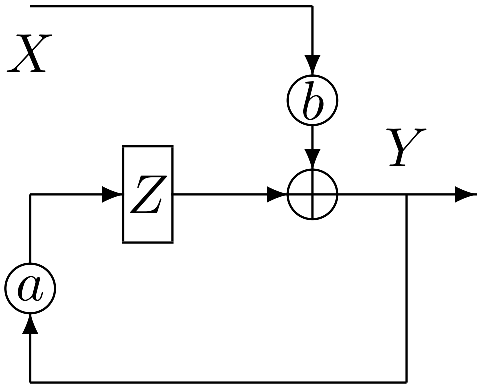
As in mathematics, intermediate equations can be introduced:

```
S = X -> I; I = pre S + X
```

A classical model of control theory and electronics

Example: a linear filter

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



The idea of Lustre:

- directly write mathematical equations
- analyze, transform and simulate them
- automatically translate them into executable programs

The expressiveness of Lustre

- First order functional language managing streams, no recursion.
- Types are declared; no polymorphism; no control-structures; limited clock calculus.

Increase its expressiveness:

- Modularity (libraries), abstraction mechanisms.
- Polymorphism; type and clock inference.
- Control structures; imperative features (but in a safe way).

We started working on these questions with Paul Caspi in 1995 and introduced the class **Synchronous Kahn Networks** [ICFP'96].

Lucid Synchrone

Try to mix all the best of these two paradigms:

- Synchronous data-flow languages (Lustre).
- General purpose ML languages (Objective Caml, Haskell,...).

A language combining:

- **Synchronous data-flow** as a way to deal with time.
- **Features from ML** to increase expressiveness: E.g., type inference, polymorphism, higher-order.

Follow a few principles

- The synchronous property is checked by a dedicated type system called the **clock calculus**. Inferred clocks express static constraints on synchronization.
- Clocks are used to give a precise semantics to all programming constructs.
- Several other type-based analysis (e.g., initialisation, causality).

Lucid Synchrone

Build a “laboratory” language

- study the extensions of Lustre and SCADE (synchronous and functional)
- experiment things and write programs!
- Version 1 (1995), Version 2 (2001), V3 (2006)
- `http://www.di.ens.fr/~pouzet/lucid-synchrone`

ReLuC and SCADE 6 at Esterel-Tech.

In 2000, Esterel-Tech. was considering designing a new version of SCADE. We started a close colaboration with the compilation team.

- Several features were implemented in the ReLuC prototype compiler (`merge` instead of `current`, clock calculus, compilation into clocked equations).
- New results developped jointly: initialization analysis, hierarchical automata, etc.

This made the basis of SCADE 6 available since 2008.

Main results since 1996

- Synchronous Kahn networks [ICFP'96]
- Clocks as dependent types [ICFP'96]
- Modular compilation (co-induction *vs* co-iteration) [CMCS'98]
- ML-like clock calculus [Emsoft'03]
- causality analysis [ESOP'01]
- initialization analysis [SLAP'03, STTT'04]
- higher-order and typing [Emsoft'04]
- data-flow and state machines [Emsoft'05, Emsoft'06]
- N-Synchronous Kahn Networks [Emsoft'05, POPL'06, APLAS'08, MPC'10]
- Clock-directed code generation of synchronous data-flow [LCTES'08]
- Modular Static Scheduling [Emsoft'09, JDAES'10]

Some examples (V3)

- `int` denote the type of streams of integers,
- `1` denotes an (infinite) constant stream of 1,
- usual primitives apply point-wise

`c`	t	f	t	$\ldots$
`x`	x_0	x_1	x_2	$\ldots$
`y`	y_0	y_1	y_2	$\ldots$
`if c then x else y`	x_0	y_1	x_2	$\ldots$

Combinatorial functions

Example: 1-bit adder

```
let xor x y = (x & not (y)) or (not x & y)

let full_add(a, b, c) = (s, co)
  where
      s = (a xor b) xor c
  and co = (a & b) or (b & c) or (a & c)
```

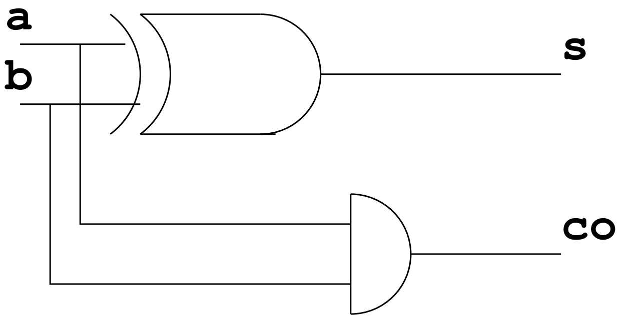
The compiler automatically computes the type and clock signature.

```
val full_add :  bool * bool * bool -> bool * bool
val full_add :: 'a * 'a * 'a -> 'a * 'a
```

Full Adder (hierarchical)

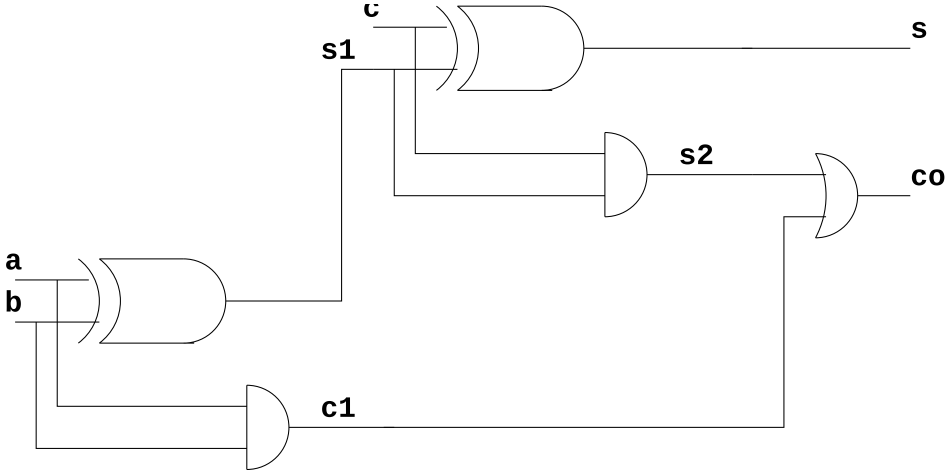
Compose two “half adder”

```
let half_add(a,b) = (s, co)
   where
       s = a xor b
   and co = a & b
```



Instanciate twice

```
let full_add(a,b,c) = (s, co)
  where
  rec (s1, c1) = half_add(a,b)
  and (s, c2) = half_add(c, s1)
  and co = c1 or c2
```



Sequential Functions

Operators `fby`, `->`, `pre`

- `fby`: unitary (initialized) delay
- `->`: initialization
- `pre`: un-initialized delay (register in circuits)

`x`	x_0	x_1	x_2	x_3	x_4	x_5	$\ldots$
`y`	y_0	y_1	y_2	y_3	y_4	y_5	$\ldots$
`x fby y`	x_0	y_0	y_1	y_2	y_3	y_4	$\ldots$
`pre x`	nil	x_0	x_1	x_2	x_3	x_4	$\ldots$
`x -> y`	x_0	y_1	y_2	y_3	y_4	y_5	$\ldots$

Warning: these operators applied to discrete signals only.

Sequential Functions

- Stream functions may depend on the past (statefull systems)
- Example: edge front detector

```
let node edge x = x -> not (pre x) & x


val edge : bool => bool
val edge :: 'a -> 'a
```

x	t	f	t	t	t	f	$\ldots$
edge x	t	f	t	f	f	f	$\ldots$

As in ML, it is also possible to give types explicitely:

```
let node edge (x:bool) = (o:bool) where
  rec o = x -> not (pre x) & x
```

In V3, we distinguish combinatorial function (`->`) from sequential functions (`=>`)

Polymorphism (code reuse)

```
let node delay x = x -> pre x

val delay : 'a => 'a
val delay :: 'a -> 'a

let node edge x = false -> x <> pre x

val edge : 'a => 'a
val edge :: 'a -> 'a
```

In Lustre, polymorphism is limited to a set of predefined operators (e.g., `if/then/else`, `when`) and does not pass abstraction.

Library and Curryfication

```
(* module Numerical *)
let node integr h x0 x' = x where
  rec x = x0 -> pre x +. x' *. h

val integr : float -> float -> float => float
val integr :: 'a -> 'a -> 'a -> 'a

(* module Main *)
let dt = 0.001
let integr0 = integr dt

val integr0 : float -> float => float
val integr0 :: 'a -> 'a -> 'a
```

Programming with equations

```
let node min_max 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

val min_max : int -> int * int
val min_max :: 'a -> 'a * 'a

let node min_max x = (min, max) where
  rec (min, max) =
        (x, x) -> if x < pre min then (x, pre max)
                  else if x > pre max then (pre min, x)
                  else (pre min, pre max)
```

Causality Analysis

Reject programs which cannot be executed sequentially.

```
let node min_max x = (min, max) where
  rec min = x -> if x < pre min then x else min
                                           ^^^^^

  and max = x -> if x > pre max then x else pre max


Error: min depends instantaneously on itself
```

- A "syntactical" criteria: a recursion must cross a delay.
- A type system (with Pascal Cuoq [ESOP'01]).
- Type signatures (interfaces) can express dependences between inputs/outputs.
- Higher-order make the analysis quite difficult.

Initialization Analysis

Reject programs for which the result depend on the initial value of some delays.

```
let node min_max x = (min, max) where
  rec min = if x < pre min then x else pre min
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  and max = x -> if x > pre max then x else pre max


Error: this expression may not be initialized
```

- Mostly a **1-bit abstraction**: a stream is either defined at every instant or possibly not at the very first only.
- A type system (with a sub-typing rule), with JL-Colaço from Esterel-Technologies [SLAP'02, STTT'04].
- It worked (surprisingly) well for SCADE. Tested on real-size examples (75000 lines) at Esterel-Tech in the ReLuC compiler (2003). Now integrated to SCADE 6.

Clocks: mix several time-scale

Mix slow and fast processes:

- E.g., multi-sampled systems (software), multi-clock (hardware).
- Filtering is not necessarily periodic: filtering can be done according to **any** boolean condition.
- How to mix slow and fast processes in a safe way?

The clock calculus:

- The clock of a stream defines the instants where a value is present (that is, available).
- The clock calculus is a dedicated type system which check that the actual clock of a stream equals the expected clock.
- In **Lucid Synchrone**, a clock is a type and is automatically inferred.

Two operators

`when` (under-sampling) and `merge` (over-sampling)

`c`	t	t	f	f	t	f	$\ldots$
`x`	x_0	x_1	x_2	x_3	x_4	x_5	$\ldots$
`x when c`	x_0	x_1			x_4		$\ldots$
`x whenot c`			x_2	x_3		x_5	$\ldots$
`y`	y_0	y_1			y_2		$\ldots$
`merge c y (x whenot c)`	y_0	y_1	x_2	x_3	y_2	x_5	$\ldots$

Clocks defined at top-level

```
let node sum x = s where rec s = x -> pre s + x
let node sampled_sum x c = sum (x when c)

val sampled_sum : int -> bool => int
val sampled_sum :: 'a -> (_c0:'a) -> 'a on _c0

let clock ten = count 10 true
let node sum_ten x = sampled_sum x ten

val ten : bool
val ten :: 'a
val sum_ten : int => int
val sum_ten :: 'a -> 'a on ten
```

```
let node hold ydef c x = y
   where rec y = merge c x ((ydef -> pre y) whenot c)

val hold : 'a -> bool -> 'a => 'a
val hold :: 'a -> (_c0:'a) -> 'a on _c0 -> 'a
```

`c`	f	t	f	f	f	t	$\ldots$
`x`		x_0				x_1	$\ldots$
`y`	y_0		y_1	y_2	y_3		$\ldots$
`ydef`	d_0	d_1	d_2	d_3	d_4	d_5	$\ldots$
`hold c x ydef`	d_0	x_0	x_0	x_0	x_0	x_1	$\ldots$

For example, `hold 0 ten` is a stuttering function.

Filtering an input *vs* filtering an output

Clocks provide a way to define control structures, that is, pieces of code which are executed according to some condition.

`x`	x_0	x_1	x_2	x_3	x_4	$\dots$
`x when c`		x_1		x_3		$\dots$
`pre x`	nil	x_0	x_1	x_2	x_3	$\dots$
`pre (x when c)`		nil		x_1		$\dots$
`(pre x) when c`		x_0		x_2		$\dots$

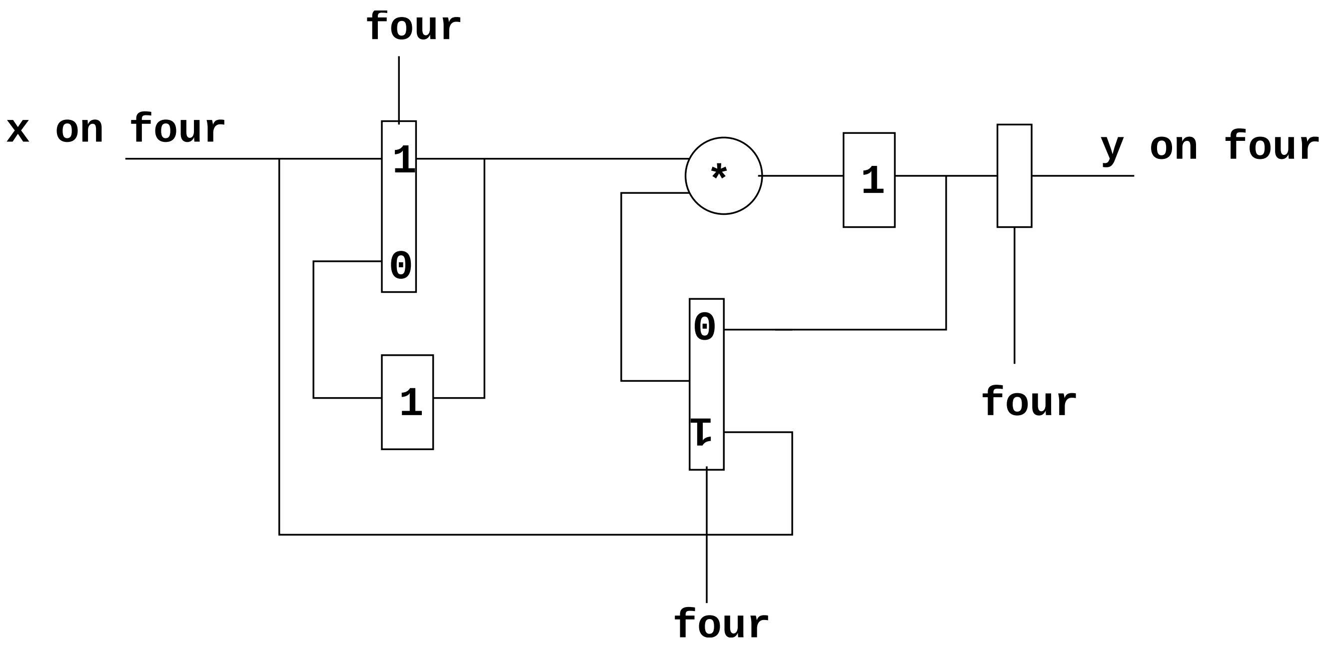
As soon as a function f is sequential, $f(x \texttt{ when } c) \neq (f(x)) \texttt{ when } c$.

Over-sampling

- Define systems whose internal rate is faster that the rate of their inputs?
- Express temporal constraints, scheduling, resources.

Example: Computation of `x^5`

```
let node power x = x * x * x * x * x


let clock four = count 4 true
let node spower x = y where
  rec i = merge four x ((1 fby i) whenot four)
  and o = 1 fby (i * merge four x (o whenot four))
  and y = o when four

val power  :: 'a -> 'a
val spower :: 'a on four -> 'a on four
```



four	t	f	f	f	t	f	f	f	t	f	f	$\ldots$
x	x_0				x_1				x_2			$\ldots$
i	x_0	x_0	x_0	x_0	x_1	x_1	x_1	x_1	x_2	x_2	x_2	$\ldots$
o	1	x_0^2	x_0^3	x_0^4	x_0^5	x_1^2	x_1^3	x_1^4	x_1^5	x_2^2	x_2^3	$\ldots$
spower x	1				x_0^5				x_1^5			$\ldots$
power x	x_0^5				x_1^5				x_2^5			$\ldots$

Property: `1 fby (power x)` and `spower x` are observationally equivalent

Nesting clocks

```
let clock sixty = sample 60


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


val hour_minute_second : 'a => 'a * 'a * 'a
val hour_minute_second :: 'a -> 'a on sixty on sixty * 'a on sixty * 'a
```

A stream on `'a on sixty on sixty` is only present one instant over 3600 instants.

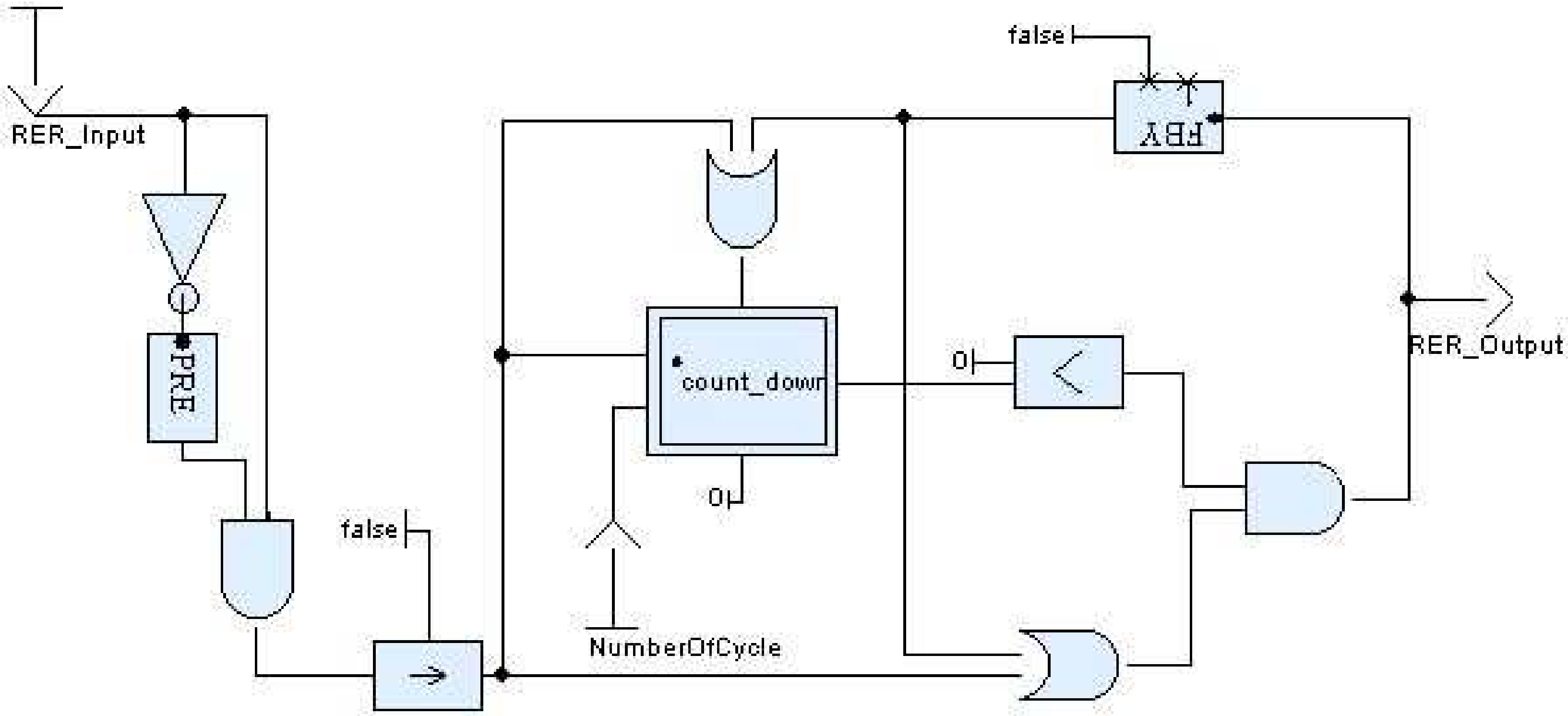
Treatment of periodic clocks:

- No particular treatment of periods. Thus, `'a on (60) on (60)` and `'a on (3600)` are considered different.
- The theory of N-synchrony allow to deal with ultimately periodic clocks: [POPL'96, APLAS'08, MPC'10].

Filtering according to some boolean condition

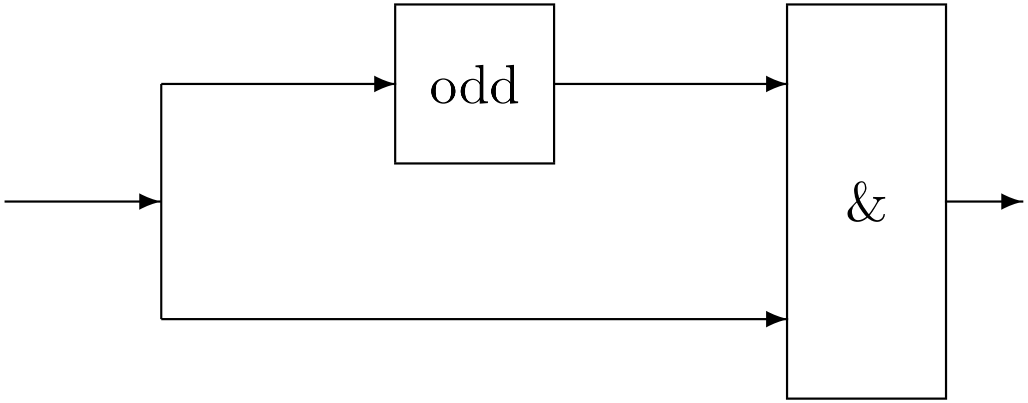
Clocks are not necessarily periodic. It is possible to filter according to any boolean condition.

E.g., the rising edge retrigger of the SCADE standard library.



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

let node rising_edge_retrigger rer_input number_of_cycle = rer_output
  where
    rec rer_output = (0 < v) & clk
    and v = merge clk (count_down ((count,number_of_cycle) 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))
```

Clock Constraints and Synchrony



The computation of $(x_n \,\&\, x_{2n})_{n \in I\!N}$ is not real-time

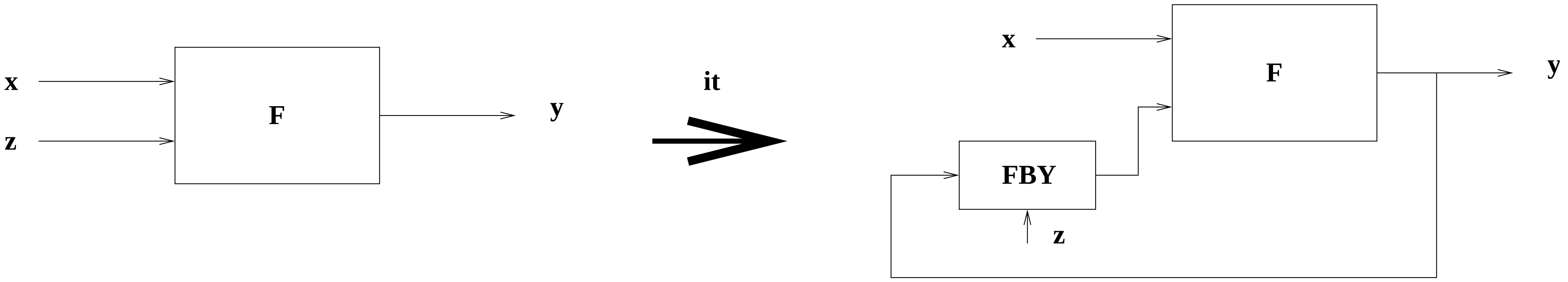
```
let odd x = x when half
let non_synchronous x =  x & (odd  x)
                             ^^^^^^^^^

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

Execution with unbounded FIFOs!!!

- clocks = an information about the behavior of streams
- clocks = types
- the `merge` and type based clock calculus is reused in the ReLuC compiler of SCADE

Higher-order

Iteration:



```
let node it f z x = y
    where rec y = f x (init fby y)
```

```
val it : ('b -> 'a -> 'a) -> 'a -> 'b => 'a
val it :: ('b -> 'a -> 'a) -> 'a -> 'b -> 'a
```

Then:

```
let node sum x = it (+) 0 x
let node mult x = it (*) 1 x
```

A word on compilation

Compiler organisation:

- Type inference then clock inference.
- At then end of these processes, every expression is annotated with its type and clock.
- Causality and initialization analysis.
- Every higher-level programming constructs (control-structures, automata, signals) are translated into the basic clocked language.

Clock-directed code generation: [LCTES'08]

- The clock serves as a guard: a variable is **only** computed when its clock is true.
- Expressions with the same clock are gathered as much as possible while respecting data-dependences.

Language extensions

This basic calculus can be extended with various features.

- Pattern matching, conditionals.
- Hierarchical automata, signals, etc.
- Everything can be translated into the basic language. Still, the code generation does not have to be redone.

Delays: `pre`, `next` and `last`

LUSTRE and LUCID SYNCHRONE are based on the unitary delay `pre` and the initialization operator `->`. `fby` is the initialized delay.

```
let node edge x = x -> not (pre x) & x
```

- If e is a signal, `pre`(e) is the value of e, the last time e has been observed.
- `pre`(e) stands for a local memory. e can be any expression.
- Thus, `pre`(x) is not necessarily the previous value of x !

```
let node f(x) = o where
  rec match x with
      | true -> do o = 0 -> pre o + 1 done
      | false -> do o = 0 -> pre o - 1 done
      end
```

x	`true`	`true`	`true`	`false`	`true`	`false`	`false`	`false`	...
o	0	1	2	0	3	1	2	3	...

The operator `last`

- If x is a signal, `last`(x) defines the value of x, the last time x was computed.
- `last`(x) is the last computed value of x
- It only applies to a name, not an expression.

```
let node f(x) = o where
  rec last o = 0 (* initialization *)
  and match x with
      | true -> do o = last o + 1 done
      | false -> do o = last o - 1 done
      end
```

x	true	true	true	false	true	false	false	false	...
o	1	2	3	2	3	2	1	0	...
`last`(o)	0	1	2	3	2	3	2	1	...

`pre` and `last`

None of the two is better than the other: `pre`/`->` can be translated into `last`/initialization and conversely.

The Lucid Synchrone compiler translates programs into a data-flow kernel using only `pre` and `->`. The new Hybrid language also.

The previous program is thus a short-cut for:

```
let node f(x) = o where
  rec l_o = 0 -> pre o (* initialization *)
  and match x with
      | true -> do o = l_o + 1 done
      | false -> do o = l_o - 1 done
      end
```

Control structures are a special form of `merge/when`

Again, the precise semantics of the previous programs can be given in term of clocked sequences.

```
let node f(x) = o where
  rec l_o = 0 -> pre o
  and o = merge x ((l_o when x) + 1) ((l_o whenot x) + 1)
```

Note that the semantics is very different for the first program:

```
let node f(x) = o where
  rec o = merge x (0 -> pre(o when x) + 1) (0 -> pre(o whenot x) + 1)
```

In the first case, we access `(0 -> pre o) when x`.

In the second, we access `0 -> pre(o when x) + 1`.

A remark on `next` *versus* `pre`

Let T be a discrete set of instants $T = \{t_0, ..., t_n, ...\}$ and two signals $x : T \mapsto V$, $y : T \mapsto V$. Then:

- `pre`$(x)(t_n) = x(t_{n-1})$ and `pre`$(x)(t_0) = nil$ where $nil \in V$.
- $(x$ `->` $y)(t_0) = x(t_0)$ and $(x$ `->` $y)(t_n) = y(t_n)$
- The equation:

  ```
  next z = x init y
  ```

  defines the signal $z : T \mapsto V$ such that $z(t_{n+1}) = x(t_n)$ and $z(t_0) = y(t_0)$.

Thus, any equation of this form is equivalent to:

```
next_z = x and z = y -> pre next_z
```

- None of the two is better than the other. It is mainly a matter of taste.
- Mixing both styles is confusing.
- `pre`, `->` and `last` combine quite well.

Extending Synchronous Data-flow with Automata [EMSOFT05,EMSOFT06]

Basis

- *Mode-Automata* by Maraninchi & Rémond [ESOP98, SCP03]
- *SignalGTI* (Rutten [EuroMicro95] and *Lucid Synchrone V2* (Hamon & Pouzet [PPDP00, SLAP04])

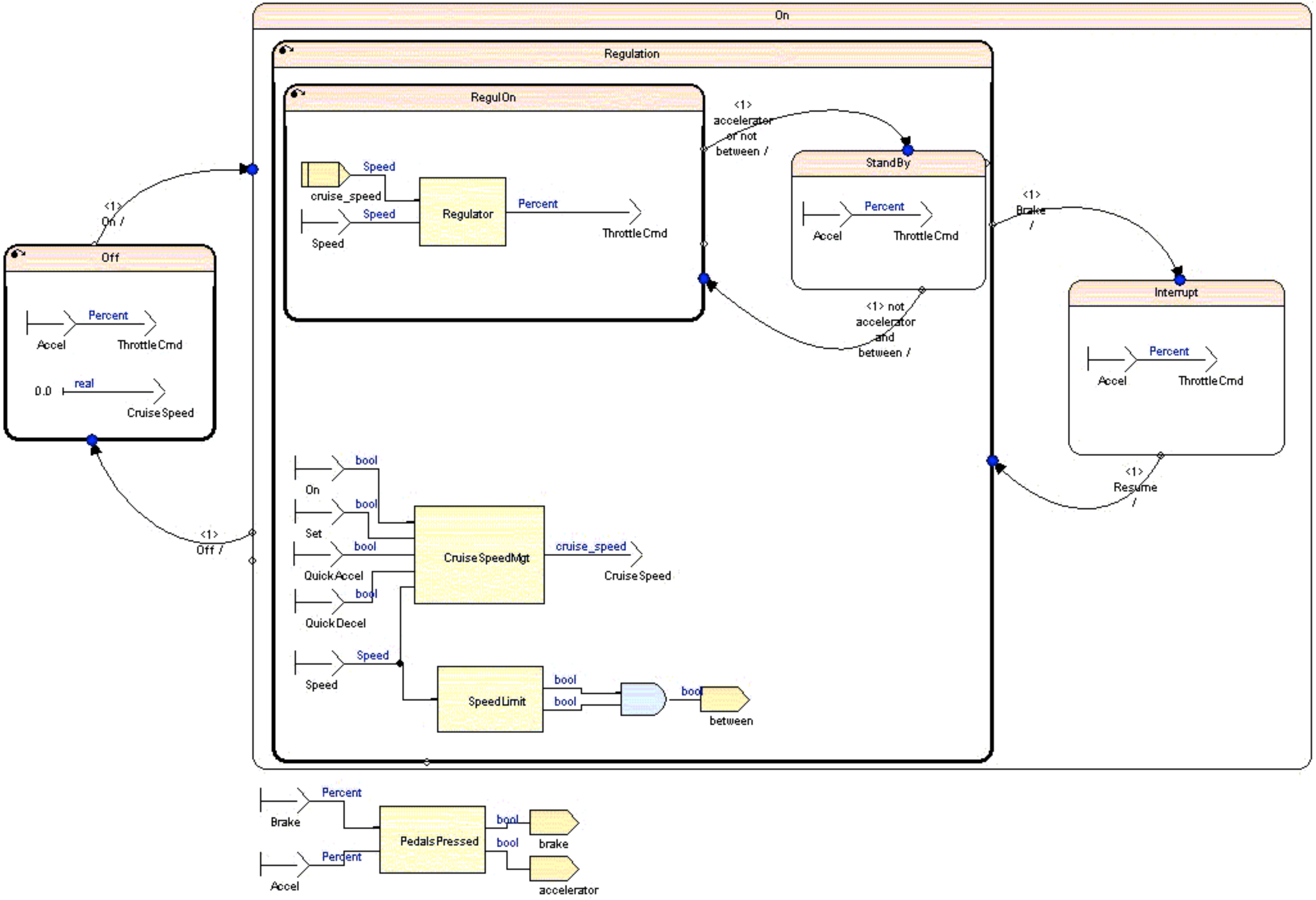
Proposal

- Extend a basic clocked calculus (SCADE/Lustre) with automata constructions.
- Base it on a *translation semantics* into well clocked programs; gives both the semantics and the compilation method.

Two implementations

- *Lucid Synchrone* language and compiler
- *ReLuC* compiler of SCADE at Esterel-Technologies; the basis of SCADE V6 (released in 2008)

The Cruise Control with SCADE 6

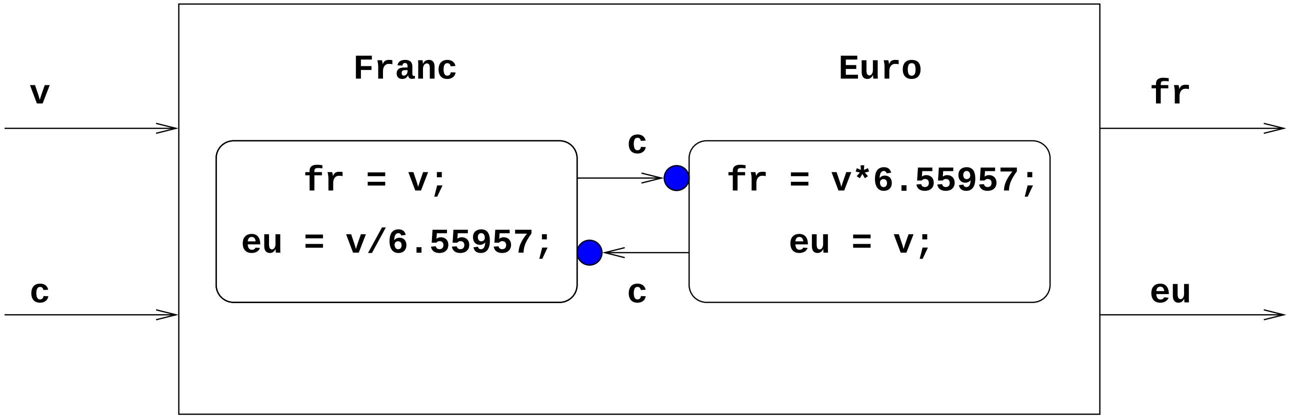


Semantic principles

- only one set of equations is executed during a reaction
- two kinds of transitions: Weak delayed ("until") or Strong ("unless")

- both can be "by history" (H* in UML) or not (if not, both the SSM and the data-flow in the target state are reseted
- at most one strong transition followed by a weak transition can be fired during a reaction
- at every instant:
  - what is the current active state?
  - execute the corresponding set of equations
  - what is the next state?
- forbids arbitrary long state traversal, simplifies program analysis, better generated code

An example: the Franc/Euro converter



in *Lucid Synchrone* syntax:

```
let node converter v c = (euro, fr) where
  automaton
  | Franc -> do fr = v and eur = v / 6.55957
            until c then Euro
  | Euro -> do fr = v * 6.55957 and eu = v
           until c then Franc
  end
```

Remark: `fr` and `eur` are *shared flow* but with only one definition at a time

Strong vs Weak pre-emption

Two types of transitions can be considered

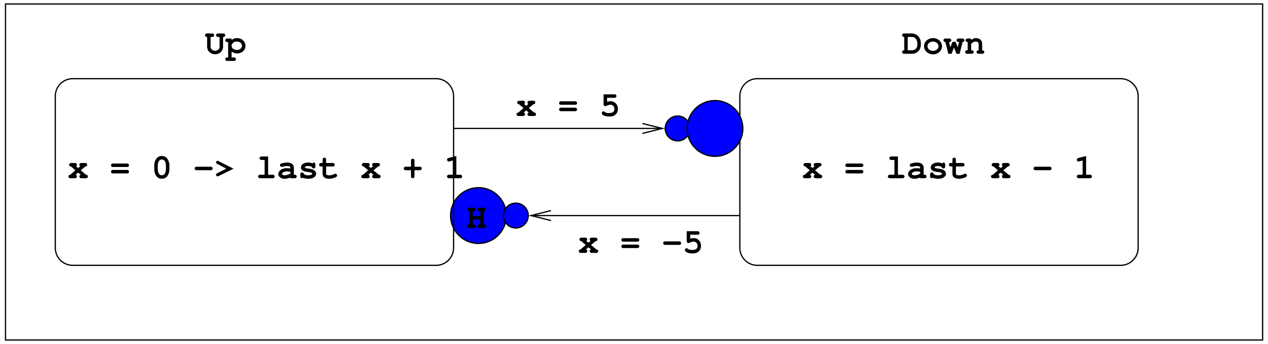
```
let node converter v c = (euro, fr) where
  automaton
  | Franc -> do fr = v and eur = v / 6.55957
             unless c then Euro
  | Euro -> do fr = v * 6.55957 and eu = v
            unless c then Franc
  end
```

- `until` means that the escape condition is executed after the body has been executed
- `unless` means that the escape condition is executed before and determines the active state of the reaction

Equations and Expressions in States

- every state defines the current value of a *shared flow*
- a flow must be defined only once per cycle
- the Lustre “pre” is local to its upper state (`pre e` gives the previous value of `e`, the last time `e` was alive)
- the substitution principle of Lustre is still true at a given hierarchy ⇒ data-flow diagrams make sense!
- the notation `last x` gives access to the latest value of `x` in its scope.
- an absent definition for a shared flow `x` is implicitly complemented (i.e., `x = last x`)

Mode Automata, a simple example



```
x = 0 1 2 3 4 5 4 3 2 1 0 -1 -2 -3 -4 -5 -4 -3 -2 -1 0 ...
```

```
let node two_modes () = x where
  rec automaton
      | Up -> do x = 0 -> last x + 1
              until x = 5 continue Down
      | Down -> do x = last x - 1
                until x = -5 continue Up
      end
```

Remark: replacing `until` by `unless` would lead to a causality error!

Implicit completion of absent definitions

```
let node modes up down init = o where
  automaton
  | Await -> do o = init then Up
  | Counting -> do automaton
                    Up -> do o = last o + 1 unless down then Down
                  | Down -> do o = last o - 1 unless up then Up
                  end
              unless up & down then Silent
  | Silent -> do then Up
  end
```

- `do ... then Up` is a short-cut for `do ... until true then Up`
- the absent equation for `x` in the state `Silent` is implicitly `x = last x`

Translation semantics

- use clocks to give a precise semantics: we know how to compile clocked data-flow programs efficiently (cf. LUCID SYNCHRONE and RELUC compilers)
- give a translation semantics into the basic data-flow language
- type and clocks are preserved during the source-to-source transformation

Several steps

- compilation of the automaton construction into the control structures (`match` statements)
- compilation of the `reset` construction between equations into the basic reset
- elimination of shared memory `last x`

Two new features

Parameterized State Machines:

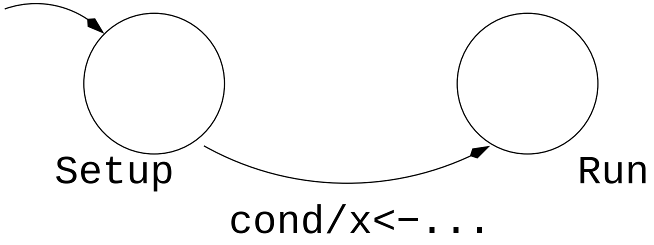
this provides a way to pass local information between two states without interfering with the rest of the code

Valued Signals:

These are events tagged with values as found in Esterel and provide an alternative to regular flows when programming control-dominated systems

Parameterized State Machines

- it is often necessary to communicate values between two states upon taking a transition
- e.g., a *setup* state communicate initialization values to a *run* state



- can we provide a safe mechanism to communicate values between two states?
- without interfering with the rest of the automaton, i.e.,
- without relying on global shared variables (and imperative modifications) in states nor transitions?

Parameterized states:

- states can be Parameterized by initial values which can be used in turn in the target automaton
- preserves all the properties of the basic automata

A typical example

several modes of normal execution and a failure mode which needs some contextual information

```
let node controller in1 in2 = out where
  automaton
  | State1 ->
     do out = f (in1, in2)
     until (out > 10) then State2
     until (in2 = 0) then Fail_safe(1, 0)
  | State2 ->
     let rec x = 0 -> (pre x) + 1 in
     do out = g (in1,x)
     until (out > 1000) then Fail_safe(2, x)
  | Fail_safe(error_code, resume_after) ->
     let rec resume = resume_after -> (pre resume) - 1 in
     do out = if (error_code = 1) then 0 else 1000
     until (resume <= 0) then State2
  end
```

Valued Signals and Signal Pattern Matching

- in a control structure (e.g., automaton), every shared flow must have a value at every instant
- if an equation for `x` is missing, it keeps implicitly its last value (i.e., `x = last x` is added)
- how to talk about absent value? If `x` is not produced, we want it to be absent
- in imperative formalisms (e.g., Esterel), an event is present if it is explicitly emitted and considered absent otherwise
- can we provide a simple way to achieve the same in the context of data-flow programming?

An example

```
let node sum x y = o where
  present
  | x(v) & y(w) -> do o = v + w done
  | x(v1) -> do o = v1 done
  | y(v2) -> do o = v2 done
  | _ -> do o = 0 done
  end

val sum : int sig -> int sig => int
val sum :: 'a sig -> 'a sig -> 'a
```

Accessing the value of a valued signal

- the value of a signal is the one which is emitted during the reaction
- what is the value in case where no value is emitted?
- **Esterel:** keeps the last computed value (i.e., implicitly complement the value with a register)

  ```
  emit S( ?A + 1)
  ```

  this is **unsafe** and raises **initialization problems**: what is the value if it has never been emitted?

- need extra methodology development rules to guard every access by a test for presence

  ```
  present A then ... emit S(?A + 1) ...
  ```

Signal pattern matching

- a pattern-matching construct testing the presence of valued signals and accessing their content
- a block structure and only present value can be accessed

```
let node sum x y = o where
  present
  | x(v) & y(w) -> do emit o = v + w done
  | x(v1) -> do emit o = v1 done
  | y(v2) -> do emit o = v2 done
  | _ -> do done
  end
```

```
val sum : int sig -> int sig -> int sig
val sum :: 'a sig -> 'a sig -> 'a sig
```

Signals as Existential Types

A signal is nothing but a pair made of:

- a boolean sequence c which is itself on clock type ck
- a sequence sampled on c, that is, with clock type ck `on` c

Then, clock verification is almost trivial and can be adapted from Laufer & Oderski extension for existential types in ML.

Initialization analysis

The initialization analysis must now take into account the semantics of automata.

```
let node two x = o where
  automaton
    S1 -> do o = 0 -> last o + 1
          until x continue S2
  | S2 -> do o = last o - 1 until x continue S1
  end
```

`o` is clearly well defined. This information is hidden in the translated program.

```
let node two x = o where
  rec o = merge s (S1 -> 0 -> (pre o) when S1(s) + 1)
                  (S2 -> (pre o) when S2(s) - 1)
  and ns = merge s (S1 -> if x when S1(s) then S2 else S1)
                   (S2 -> if x when S2(s) then S1 else S2)
  and clock s = S1 -> pre ns
```

Initialisation analysis

For any variable `x` defined in an initial state only left with a weak transition, `last x` is well initialized in the remaining states.

The following program is not well initialized.

```
let node two x = o where
  automaton
  | S1 -> do o = 0 -> last o + 1
          unless x continue S2
  | S2 -> do o = last o - 1
          until x continue S1 end
```

- The reasonning is local (for each automaton).
- This is because at most two transitions are fired during a reaction (strong to weak)

This analysis is implemented in Lucid Synchrone V3 (2006) and SCADE 6.

Conclusion/Current/Future Works

Compilation, semantics

- Other extensions, program analysis, etc.
- Certified compilation (for software).

Relaxed Synchrony for Video Systems

- Deal with non strictly synchronous systems but which can be synchronized through the insertion of buffers?
- The model of N-Synchronous Kahn Networks [Emsoft'05, POPL'06, APLAS'08, MPC'10]

Hybrid Systems

- Mix discrete and continuous systems is the next important step for synchronous languages [CDC'10, LCTES'11].
- Talk this afternoon.

See current works on synchronous languages at: `http://synchronics.inria.fr`