Programming Embedded Systems with Lustre

Timothy.Bourke@inria.fr

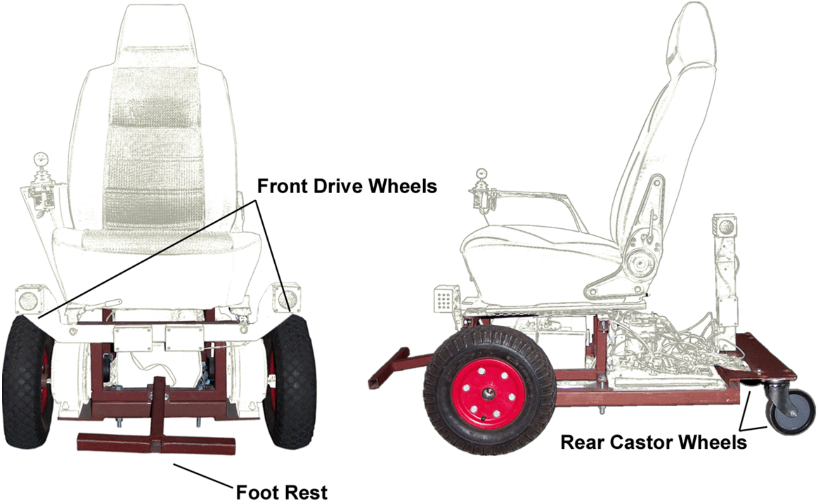
and

Marc.Pouzet@ens.psl.eu

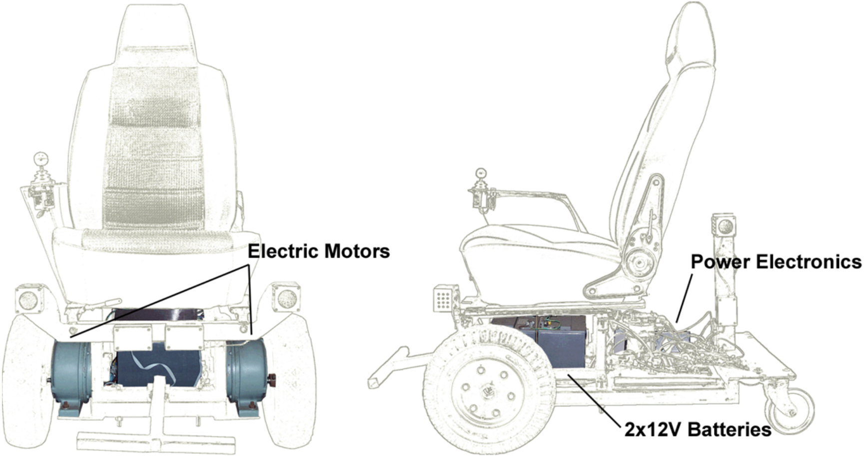
Wheelchair: An old, simple, but concrete example

- The UOW ‘robotic’ wheelchair
- Goal: low-cost mobility assistance
- Target of engineering student projects

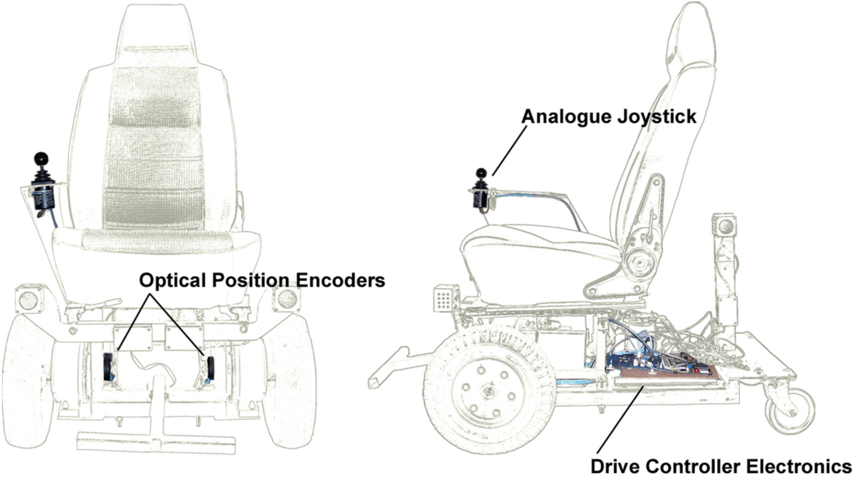
Wheelchair: mechanical structure



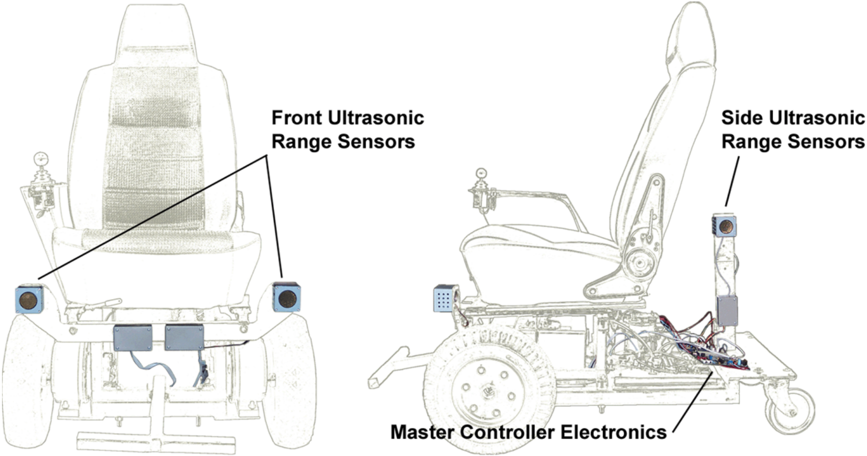
Wheelchair: power electronics



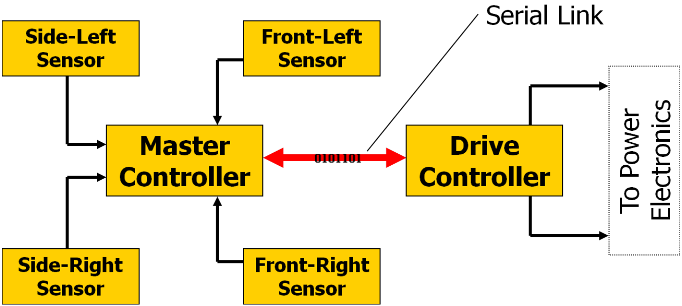
Wheelchair: drive controller



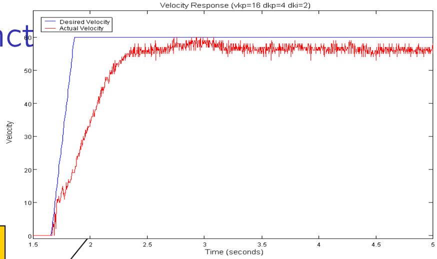
Wheelchair: master controller

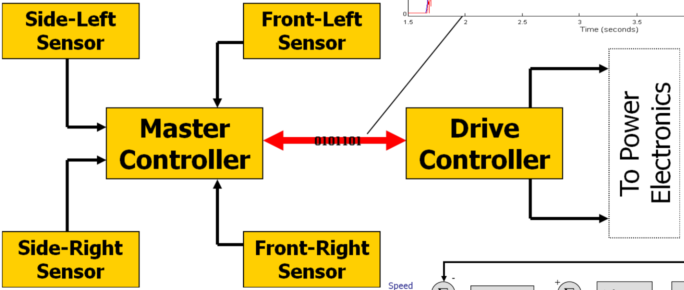


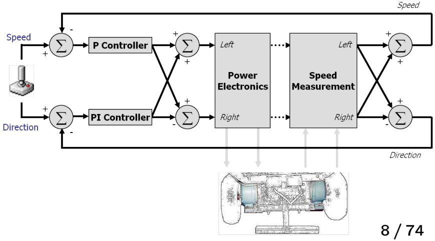
Wheelchair: architecture and functions



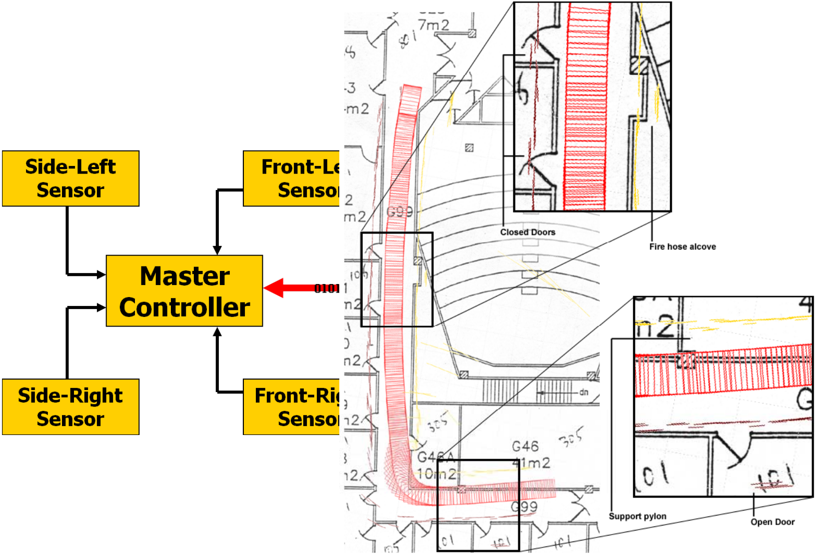
Wheelchair: architecture and funct



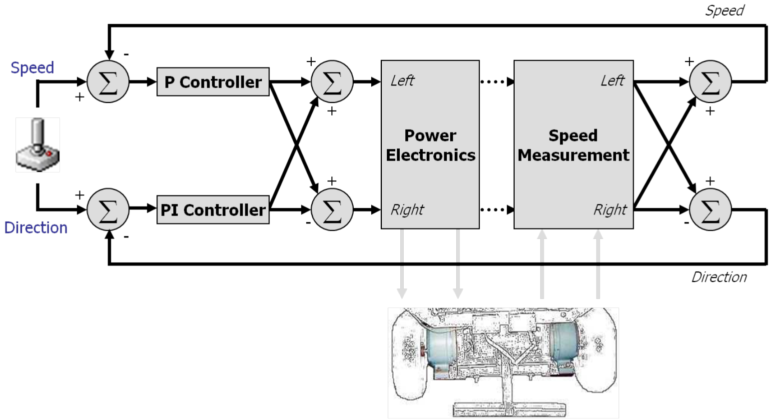




Wheelchair: architecture and functions

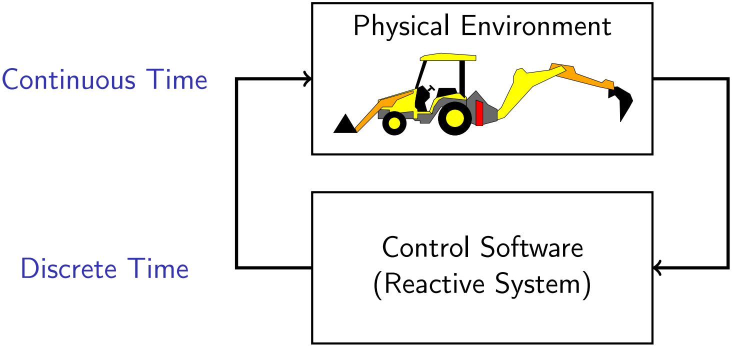


Wheelchair: drive control feedback loop



- fly-by-wire: $\approx$ 1,5MLOC
- critical software
- "dynamic" = "too late"
- cyclic execution

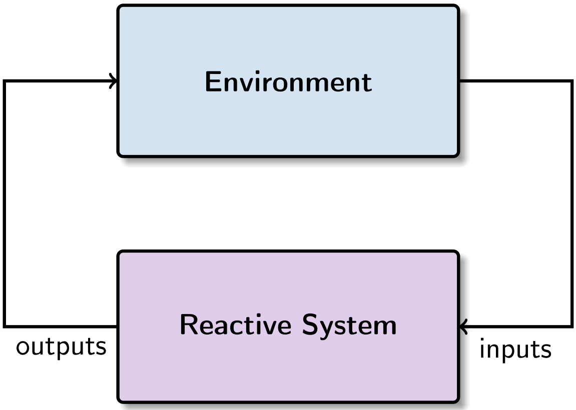
Embedded Real-Time Software



- 2005 Mars Reconnaissance Orbiter: 545 000 LoC
- 2010 Lockheed Martin F-22 Raptor: 2 500 000 LoC
- 2012 Drug infusion pump 170 000 LoC
- 2017 Primary Flight Controller A350-1000: 1 500 000 LoC generated

[Dvorak (ed.) (2009): NASA Study on Flight Software Complexity] [The Economist (2012): Open-source medical devices: When code can kill or cure]

Program and control a reactive system?



Write Assembly / C / C++ / JavaScript / Python / OCaml / Coq code by hand?

Write Assembly / C / C++ / JavaScript / Python / OCaml / Coq code by hand?

- Concurrent Designs

  Multiple OS tasks communicating via shared memory with locking to ensure atomicity?

  » Complicated implementations for complicated applications?
  » Effect of non-determinism (interleaving/preemption) on reasoning/testing?
  » What about subtle effects like deadlocks and priority inversion?

Write Assembly / C / C++ / JavaScript / Python / OCaml / Coq code by hand?

- Concurrent Designs

  Multiple OS tasks communicating via shared memory with locking to ensure atomicity?

  » Complicated implementations for complicated applications?
  » Effect of non-determinism (interleaving/preemption) on reasoning/testing?
  » What about subtle effects like deadlocks and priority inversion?

- What about lighter-weight compositions?

Write Assembly / C / C++ / JavaScript / Python / OCaml / Coq code by hand?

- Concurrent Designs

  Multiple OS tasks communicating via shared memory with locking to ensure atomicity?

  » Complicated implementations for complicated applications?
  » Effect of non-determinism (interleaving/preemption) on reasoning/testing?
  » What about subtle effects like deadlocks and priority inversion?

- What about lighter-weight compositions?

- What is the programmer's model of time?
- How are timing details expressed, implemented, validated?

Write Assembly / C / C++ / JavaScript / Python / OCaml / Coq code by hand?

- Concurrent Designs

  Multiple OS tasks communicating via shared memory with locking to ensure atomicity?

  » Complicated implementations for complicated applications?
  » Effect of non-determinism (interleaving/preemption) on reasoning/testing?
  » What about subtle effects like deadlocks and priority inversion?

- What about lighter-weight compositions?

- What is the programmer's model of time?
- How are timing details expressed, implemented, validated?

- What about writing reusable libraries?
- What about targeting different platforms?

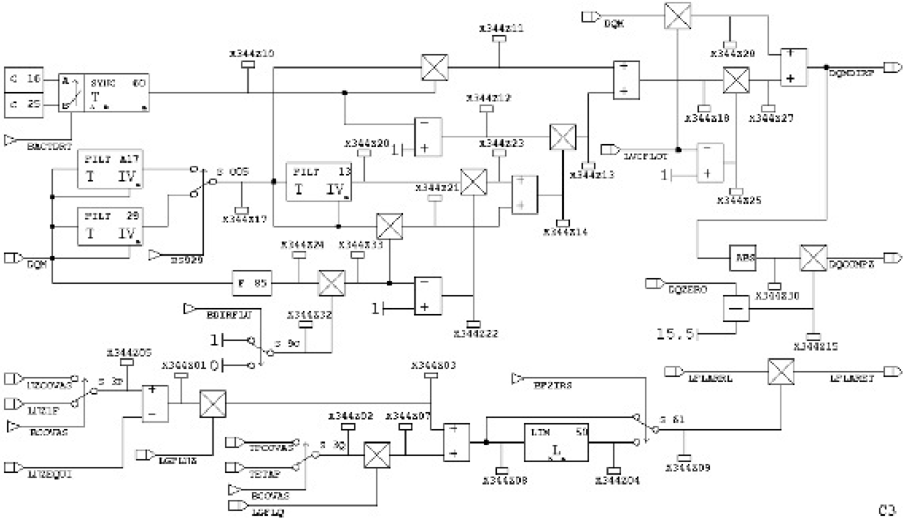
Arduino demo

1. Blinking an LED
2. Blinking two LEDs

In any case, what should the implementation be compared to?

What is the specification?

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



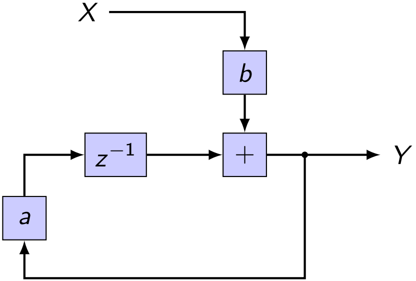
They were very precise drawings...

Control and signal engineers described control/command systems with very precise mathematics before computers were used.

Stream equations, z-transforms, (discrete) difference equations

Example: a linear filter

$$Y_0 = b \cdot X_0$$
$$\forall n,\ Y_{n+1} = a \cdot Y_n + b \cdot X_{n+1}$$



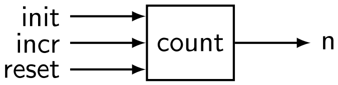
- Important idea: block diagrams can be given a precise meaning in terms of equations on stream variables.
- ...but they're not executable.
- Should we just write code and convince ourselves that it is correct?

How to make those mathematics executable?

Somewhere in Grenoble. . . the language Lustre (1984)

[Caspi, Pilaud, Halbwachs, and Plaice (1987): LUSTRE: A declarative language for programming synchronous systems] [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]

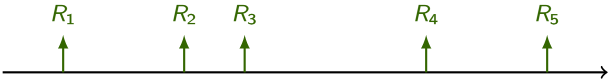
[Jahier, Raymond, and Halbwachs (2019): The Lustre V6 Reference Manual]




```
node COUNT (init, incr: int; reset: bool)
   returns (n: int);
let
   n = init ->
        if reset then init else pre(n) + incr;
tel;
```

Basic model: cyclic execution

```
every trigger:
  read inputs;
  compute;
  write outputs
```

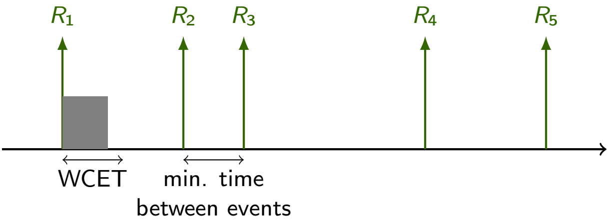


model: $R_1, R_2, R_3, R_4, R_5, \ldots$

- Often periodic with sampling, but not always.
- Reactions are atomic: buffer new inputs during `compute`.
- Assume that reactions occur at least as frequently as inputs.
- Reason in logical time rather than physical time.
  - The execution model (semantics) of a program is a sequence of reactions.
  - It abstracts from the time between reactions
    —even if physical time still matters to programmers.
- This is a significant simplification; it divides the problem in two:

1. Specify and reason about logical behaviour;
2. Separately validate assumptions on real-time behaviour.

Validating Timing Assumptions



- Calculate an upper-bound on execution time:
  Worst Case Execution Time (WCET)
- Check that WCET < minimum time between any two events
- Typical case: WCET < sample period

- The WCET is calculated on the generated assembly code for a given processor configuration (pipeline depth, cache sizes).
- Overapproximation at function calls and branching instructions.

Program by writing stream equations

A discrete-time system: a stream function; streams are synchronous.

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

The equation $Z = X + Y$ means $\forall n,\ Z_n = X_n + Y_n$.

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

For simple expressions, we effectively just added a loop around a sequence of assignment statements.

It gets more interesting when we add abstraction and state.

The idea will be to compose stream functions to build more and more sophisticated systems.

Syntax of Lustre nodes

1. Declare a block (a "node") by giving its interface:
   name, list of input variables, list of output variables

   ```
   node full_add(a : bool; b :bool; c : bool) returns (s : bool; co : bool);
   ```

   or

   ```
   node full_add(a, b, c : bool) returns (s, co : bool);
   ```

   There are three basic types: bool, int, real/float.

2. The interface is optionally followed by a list of local variables.

   ```
   var t1, t2 : bool; t3 : int;
   ```

3. The relation between inputs and outputs is defined by a list of equations. Each output and local variable must be defined exactly once.

   ```
   let
     s = e1;
     co = e2;
     (t1, t2, t3) = e3;
   tel;
   ```

   The order of the equations is not significant.

Combinatorial expressions

- Boolean operators

  not, and, or, xor

- Comparisons

  <, <=, =, <>, >=, >

- Arithmetic

  + − ∗ / div mod
  +. −. ∗. /.

- Conversions

  int, float

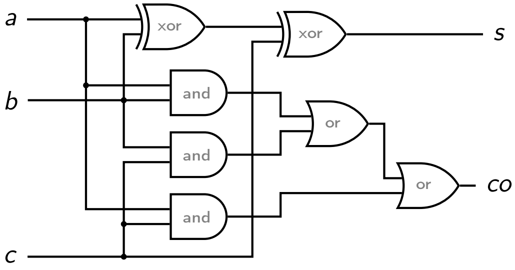
- Multiplexing

  if e1 then e2 else e3

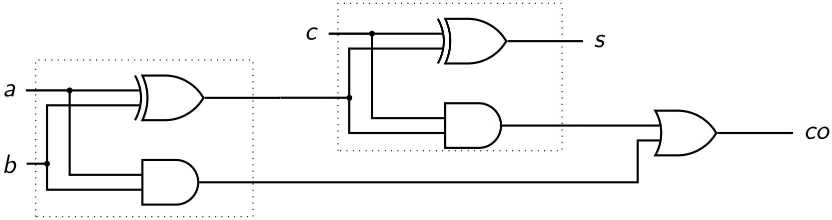
Exercise: programming 1-bit adders

Full adder (`full_add`)

a	b	c	s	co
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



Composition of two half adders (`full_add_h`)



Simulate with `make runtrace_full_add` using the provided Makefile

Exercise: model checking 1-bit adders

How to be sure that `full_add` and `full_add_h` are equivalent?

$$\forall a, b, c : \texttt{bool}.\ \texttt{full\_add}(a, b, c) = \texttt{full\_add\_h}(a, b, c)$$

Implement the following interface so that it returns `true` exactly when two full adder implementations return the same value for the same inputs.

```
-- file fulladder.lus
node equivalence(a, b, c : bool) returns (ok : bool);
```

Use the model-checking tool `lesar` to check for all possible inputs:

```
% lesar fulladder.lus equivalence -diag
--Pollux Version 2.2

TRUE PROPERTY
```

Type `make check` to run lesar. Try introducing a bug and rechecking.

Exercise: more combinatorial nodes

Return the maximum at each instant

```
node max(i1, i2 : int) returns (o : int);
```

Return the absolute value at each instant

```
node abs(i : int) returns (o : int);
```

Saturate a signal when it exceeds bounds

```
node saturate(lb, i, ub : int) returns (o : int);
let
  -- assert (lb <= ub);
  ...
tel;
```

- Build up a library of stream functions to write more abstract programs.
- Ideal: specify a system in Lustre, rely on the compiler to produce efficient low-level code.

The Unit Delay (pre) and Initialization Operator (->)

An operator for referring to the "previous" value of a stream.

(A register in a digital circuit; the z^{-1} operator in digital signal processing.)

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

nil represents an arbitrary value. It is *not* represented explicitly at runtime. The actual value is whatever happens to be in memory.

Calculate the running sum of values on X,
i.e., the stream $(S_n)_{n \in \mathbb{N}}$ with $S_0 = X_0$ and for $n > 0$, $S_n = S_{n-1} + X_n$.

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

Introducing intermediate equations does not change the meaning:

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

Programming with -> and pre

Rising edge detection

```
node redge (b : bool) returns (edge : bool);
let
   edge = false -> (b and not (pre b));
tel
```

Modulo counter

```
node mod_count (m : int) returns (out : int);
let
  out = (0 -> (pre out + 1)) mod m;
tel
```

Count of trues and falses?

```
node tf_count (in : bool) returns (out : int);
let
  out = if in then mod_count(512) else mod_count(512);
tel;
```

What is wrong with this program?

Exercise: programming with -> and pre

Return true if an input has always been true

```
node always(i : bool) returns (o : bool);
```

Count true values

```
node count_true(i : bool) returns (o : int);
```

Count successive true values (i.e., false = restart from zero)

```
node count_succ_true(i : bool) returns (o : int);
```

Tracking minimum and maximum bounds of a stream

```
node bounds(i : int) returns (min, max : int);
```

The Initialized Unit Delay (`fby`)

- Separating delay (`pre`) and initialization (`->`) can be useful.
- But, an extra analysis is necessary to check that `pre` is used correctly, i.e., that outputs never depend on an undefined (nil) value.
  [Colaço and Pouzet (2004): Type-based initialization analysis of a synchronous dataflow language]

- The `fby` (*followed by*) operator combines delay and initialization.

  `X fby Y` defines the same stream as `X -> (pre Y)`
- A `fby` with a constant at left is easy to compile into efficient code since the generated state variable can be initialized directly.

- `e0 -> e1` and `if (true fby false) then e0 else e1` define the same stream.

- NB: Lustre v4 does not have `fby`.

Exercise: simple filtering with `fbys` in Heptagon

- Consider the input signal:

$$x(n) = \begin{cases} (1.02)^n + 0.5\cos(2\pi n/8 + \pi/4) & \text{if } 0 \leq n \leq 40 \\ 0 & \text{otherwise} \end{cases}$$

- We want to recover the exponential component by eliminating the sinusoidal component (the noise).
- Filter with a 3-point running average:

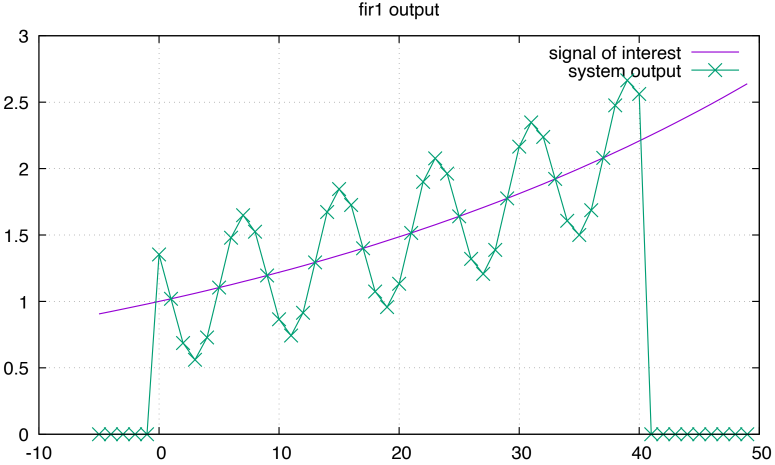
$$y(n) = \frac{1}{3}\left(\sum_{k=0}^{2} x(n-k)\right)$$

- ...and a 7-point running average:

$$y(n) = \frac{1}{7}\left(\sum_{k=0}^{6} x(n-k)\right)$$

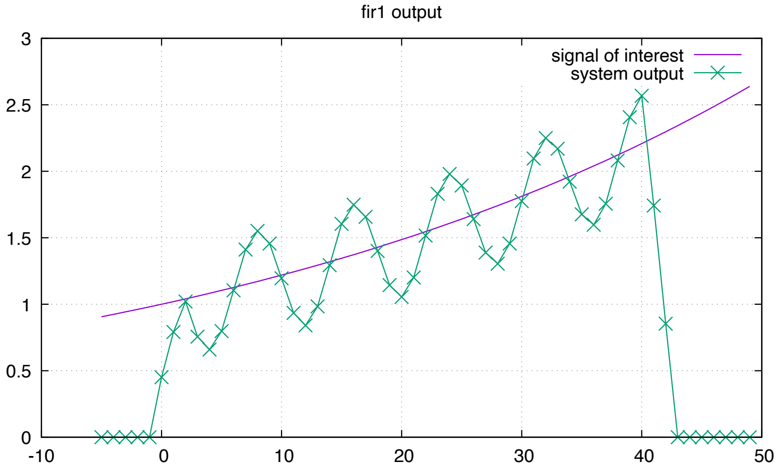
- Both are basic FIR (Finite Impulse Response) filters.
- NB: use +. and /. on values of type float.

Exercise: simple filtering with `fbys` in Heptagon



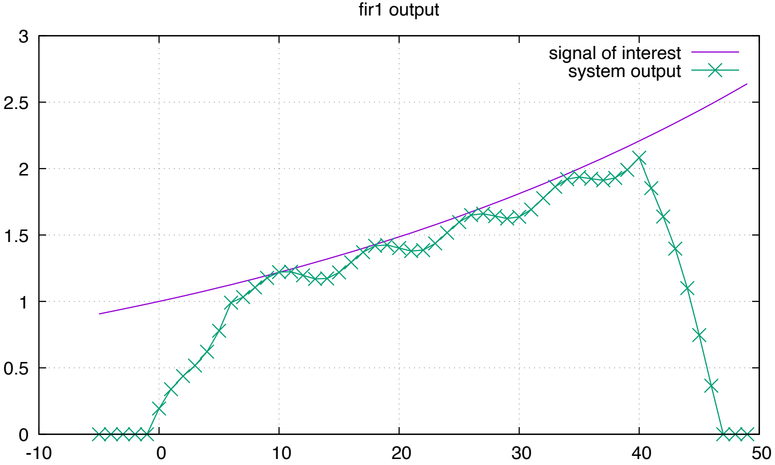
0 points

Exercise: simple filtering with `fbys` in Heptagon



3 points

Exercise: simple filtering with `fbys` in Heptagon



7 points

Cyclic Definitions

A program may have

- no solutions: `x = x + 1` or `x = y + 1 and y = x + 2`
- many solutions: `x = x` or `y = x and x = y` (nondeterminism)

Nondeterminism may be useful for abstract specifications and it induces a rich theory, i.e., different notions of equivalence in process algebra.

But, for critical embedded systems, we must guarantee at compile time that a program has exactly one solution, i.e., trace equivalence suffices.

The programmer chooses what happens, not a compiler or scheduler.

Trace equivalence simplifies testing, reasoning, and reproducibility.

Causality

The term causality refers to the relations between "events" in a program.

It's a rich subject, but for basic Lustre programs it suffices to

- Generate a dependency graph and
- Check, by topological sort, that it contains no cycles.

No dependencies are recorded for `pre` or the right-hand-side of a `fby`.

In Lustre, every dependency cycle must be broken by one or more delays.

Dependency analysis

```
node regulator (cruise_speed, speed : float; rst : bool)
returns (throttle : float);
var delta, aux : float;
let
  delta = cruise_speed - speed;
  throttle = delta * 10.0 + aux * 0.2;
  aux = if rst then delta else delta + aux;
tel
```

Dependency analysis

```
node regulator (cruise_speed, speed : float; rst : bool)⇐
returns (throttle : float);
var delta, aux : float;
let
  delta = cruise_speed − speed;
  throttle = delta ∗ 10.0 + aux ∗ 0.2;
  aux = if rst then delta else delta + aux;
tel
```

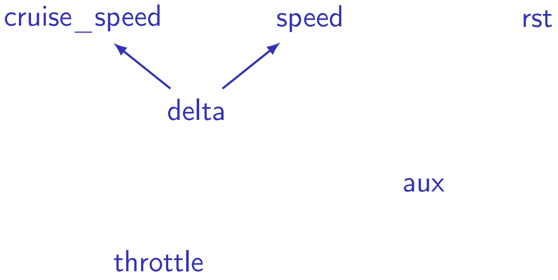
cruise_speed   speed   rst

Dependency analysis

```
node regulator (cruise_speed, speed : float; rst : bool)
returns (throttle : float);⇐
var delta, aux : float;⇐
let
  delta = cruise_speed − speed;
  throttle = delta ∗ 10.0 + aux ∗ 0.2;
  aux = if rst then delta else delta + aux;
tel
```

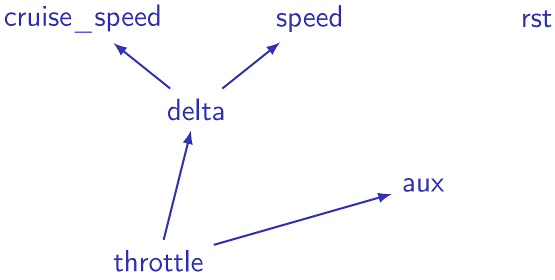
Dependency analysis

```
node regulator (cruise_speed, speed : float; rst : bool)
returns (throttle : float);
var delta, aux : float;
let
  delta = cruise_speed − speed; ⇐
  throttle = delta ∗ 10.0 + aux ∗ 0.2;
  aux = if rst then delta else delta + aux;
tel
```



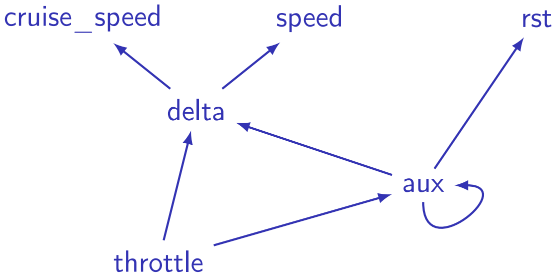
Dependency analysis

```
node regulator (cruise_speed, speed : float; rst : bool)
returns (throttle : float);
var delta, aux : float;
let
  delta = cruise_speed − speed;
  throttle = delta ∗ 10.0 + aux ∗ 0.2;  ⇐
  aux = if rst then delta else delta + aux;
tel
```



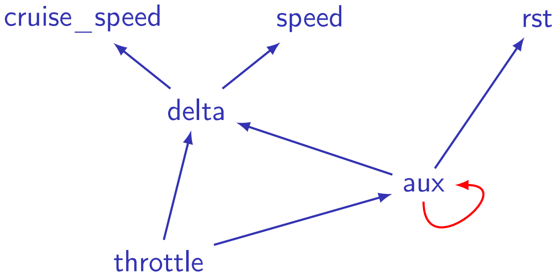
Dependency analysis

```
node regulator (cruise_speed, speed : float; rst : bool)
returns (throttle : float);
var delta, aux : float;
let
  delta = cruise_speed − speed;
  throttle = delta ∗ 10.0 + aux ∗ 0.2;
  aux = if rst then delta else delta + aux;  ⇐
tel
```



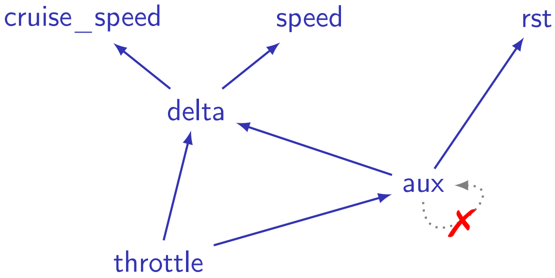
Dependency analysis

```
node regulator (cruise_speed, speed : float; rst : bool)
returns (throttle : float);
var delta, aux : float;
let
  delta = cruise_speed − speed;
  throttle = delta ∗ 10.0 + aux ∗ 0.2;
  aux = if rst then delta else delta + aux;  ⇐
tel
```



Dependency analysis

```
node regulator (cruise_speed, speed : float; rst : bool)
returns (throttle : float);
var delta, aux : float;
let
  delta = cruise_speed − speed;
  throttle = delta ∗ 10.0 + aux ∗ 0.2;
  aux = if rst then delta else delta + (0.0 −> pre aux);
tel
```



Modular causality analysis

Accept this program?

```
node direct(x : int)
returns (y : int);
let
  y = x;
tel

node main1(w : int)
returns (z : int);
let
  z = direct(z);
tel
```

Modular causality analysis

Accept this program?

```
node direct(x : int)
returns (y : int);
let
  y = x;
tel

node main1(w : int)
returns (z : int);
let
  z = direct(z);
tel
```

What about this one?

```
node delayed(x : int)
returns (y : int);
let
  y = 0 fby x;
tel

node main2(w : int)
returns (z : int);
let
  z = delayed(z);
tel
```

Modular causality analysis

Accept this program?

```
node direct(x : int)
returns (y : int);
let
  y = x;
tel

node main1(w : int)
returns (z : int);
let
  z = direct(z);
tel
```

What about this one?

```
node delayed(x : int)
returns (y : int);
let
  y = 0 fby x;
tel

node main2(w : int)
returns (z : int);
let
  z = delayed(z);
tel
```

The causality of a program can be analysed statically and modularly using a dedicated type system.

Modular causality analysis and compilation

Should this program be rejected?

```
node plumbing(x, y : int)
returns (dx, dy : int);
let
  dx = x;
  dy = y;
tel
```

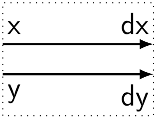
```
node main3(w : int)
returns (z : int);
var v : int;
let
  z, v = plumbing(v, w);
tel
```

Modular causality analysis and compilation

Should this program be rejected?

```
node plumbing(x, y : int)
returns (dx, dy : int);
let
  dx = x;
  dy = y;
tel
```

```
node main3(w : int)
returns (z : int);
var v : int;
let
  z, v = plumbing(v, w);
tel
```

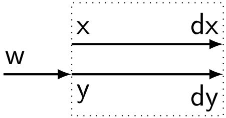


Modular causality analysis and compilation

Should this program be rejected?

```
node plumbing(x, y : int)
returns (dx, dy : int);
let
  dx = x;
  dy = y;
tel
```

```
node main3(w : int)
returns (z : int);
var v : int;
let
  z, v = plumbing(v, w);
tel
```

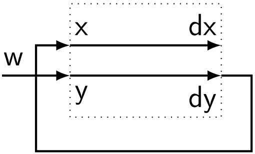


Modular causality analysis and compilation

Should this program be rejected?

```
node plumbing(x, y : int)
returns (dx, dy : int);
let
  dx = x;
  dy = y;
tel
```

```
node main3(w : int)
returns (z : int);
var v : int;
let
  z, v = plumbing(v, w);
tel
```

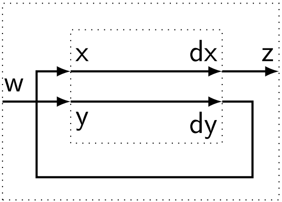


Modular causality analysis and compilation

Should this program be rejected?

```
node plumbing(x, y : int)
returns (dx, dy : int);
let
  dx = x;
  dy = y;
tel
```

```
node main3(w : int)
returns (z : int);
var v : int;
let
  z, v = plumbing(v, w);
tel
```

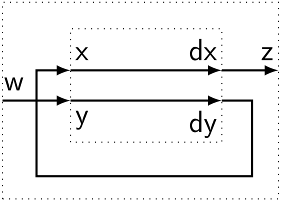


Modular causality analysis and compilation

Should this program be rejected?

```
node plumbing(x, y : int)
returns (dx, dy : int);
let
  dx = x;
  dy = y;
tel
```

```
node main3(w : int)
returns (z : int);
var v : int;
let
  z, v = plumbing(v, w);
tel
```



It depends on the compilation schema.

- Inlining and (minimize) automaton generation (Lustre v4)
  [Plaice (1988): Sémantique et compilation de LUSTRE, un langage déclaratif synchrone] [Raymond (1991): Compilation efficace d'un langage déclaratif synchrone: le générateur de code Lustre-V3]
- Modular compilation with a single step function (Scade 6/Heptagon)
  [Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]
- Modular compilation with multiple step functions
  [Pouzet and Raymond (2009): Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation] [Lublinerman, Szegedy, and Tripakis (2009): Modular Code Generation from Synchronous Block Diagrams: Modularity vs. Code Size]

Transition systems and formal specification

Transition systems are the basic formal model for reactive systems: $(S, \rightarrow)$, where $\rightarrow \subseteq S \times S$.

Mealy machines: distinguish inputs and (instantaneous) outputs: (S, S_0, I, O, T, G), where $T : S \times I \rightarrow S$ and $G : S \times I \rightarrow O$.

Many specification formalisms (e.g., I/O Automata and TLA+) essentially provide expressive languages for writing Mealy Machines.

Lustre can be viewed in a similar way...

```
automaton IncDec()
signature
  external increment,
           decrement

states
  x : Integer := 5

transitions
  external increment
  pre
    x < 10
  eff
    x := x + 1

  external decrement
  pre
    x > 0
  eff
    x := x - 1
```

Sometimes also write

```
    x' = x + 1
    next x = x + 1
```

```
automaton IncDec()
signature
  external increment,
           decrement

states
  x : Integer := 5

transitions
  external increment
  pre
    x < 10
  eff
    x := x + 1

  external decrement
  pre
    x > 0
  eff
    x := x - 1
```

Sometimes also write

```
    x' = x + 1
    next x = x + 1
```

```
node IncDec(increment, decrement : bool)
returns (nx : int);
let
  x = 5 -> pre nx;
  nx = if increment and x < 10 then x + 1
       else if decrement and x > 0 then x - 1
       else x;
tel;
```

Assuming that increment and decrement are never true at the same time.

```
node f(i0, ..., in) returns (o0, ..., om);
var s0, ..., sk, n0, ..., nk;
let
  s0, ..., sk = f_init(i0, ..., in) -> pre (n0, ..., nk);
  n0, ..., nk = f_trans(s0, ..., sk, i0, ..., in);
  o0, ..., om = f_out(s0, ..., sk, i0, ..., in);
tel
```

```
automaton IncDec()
signature
  external increment,
           decrement

states
  x : Integer := 5

transitions
  external increment
  pre
    x < 10
  eff
    x := x + 1

  external decrement
  pre
    x > 0
  eff
    x := x - 1
```

Sometimes also write

```
    x' = x + 1
    next x = x + 1
```

```
node IncDec(increment, decrement : bool)
returns (nx : int);
let
  x = 5 -> pre nx;
  nx = if increment and x < 10 then x + 1
       else if decrement and x > 0 then x - 1
       else x;
tel;
```

Assuming that `increment` and `decrement` are never true at the same time.

Lustre and other synchronous languages

- Deterministic
- Synchronous (versus asynchronous interleaving)
- Programming Languages (compile and execute)
- Composition: functions on streams
- Alternative view: iterated transition functions

Composing stream functions

Basic Lustre: a very simple language.

The idea is to build up hierarchies of functions, combining them to specify increasingly sophisticated behaviours.

With `pre`/`fby`, we can specify functions whose output is determined solely by the "history" of inputs received.

These subsystems ("blocks") can be connected together to communicate over time. A signal between two subsystems is represented as a (infinite) sequence, a "stream", of values.

Think λ-calculus but for specifying cyclic behaviours in finite memory.

Exercise: LED blinking in Lustre

Reimplement the Arduino example in Lustre.

Blink `led1` every six cycles, and `led2` every 3 cycles.

```
node main() returns (led1, led2 : bool);
```

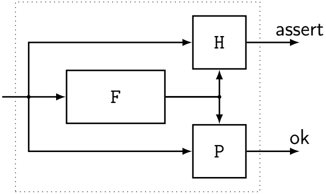
(Suggestion: break the problem down into at least three distinct nodes.)

Synchronous Observers

The comparison of programs is a particular case of a synchronous observer.

- if $y = F(x)$, we write $ok = P(x, y)$ for the property relating x and y
- and assert($H(x, y)$) to states an hypothesis on the environment.

```
node check(x:t) returns (ok:bool);
let
  assert H(x,y);
  y = F(x);
  ok = P(x,y);
tel;
```



If *assert* remains indefinitely true then *ok* remains indefinitely true always(*assert*) $\Rightarrow$ always(*ok*).

Any temporal safety property can be expressed as a Lustre program. No need to introduce a temporal logic in the language [Halbwachs, Lagnier, and Raymond (1993): Synchronous observers and the verification of reactive systems] [Halbwachs, Lagnier, and Ratel (1992): Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE];

Temporal properties are regular Lustre programs

Example of Temporal Properties

- "A is never true twice in a row": `never_twice(A)` where:

```
node never_twice(A : bool) returns (OK : bool);
let
  OK = true -> not(A and pre A);
tel;
```

- "Any event A is followed by an event B before C happens":
  followed_by(A, B) and followed_by(B, C)
  where:

```
node implies(A, B : bool) returns (OK : bool);
let
  OK = not(A) or B;
tel;

node once(A : bool) returns (OK : bool);
let
  OK = A -> A or pre OK;
tel;
```

```
node followed_by(A, B : bool)
returns (OK : bool);
let
  OK = implies(B, once(A));
tel;
```

Example of Temporal Properties (cont.)

Note: Several properties have a sequential nature, e.g., "The temperature should increase for at most 1 min or until the event stop occurs then it must decrease for 2 min".

They can be expressed as regular expressions and then translated into Lustre [Raymond (1996): Recognizing regular expressions by means of dataflow networks]

This is the basis of the language Lutin [Raymond, Roux, and Jahier (2008): Lutin: A Language for Specifying and Executing Reactive Scenarios]

For an encoding of past-time Linear Temporal Logic (LTL) see: [Halbwachs, Fernandez, and Bouajjani (1993): An executable temporal logic to express safety properties and its connection with the language Lustre]

Exercise: implementing temporal properties

1. Returns false until A occurs, then returns true from the subsequent instant onward

   ```
   node after(a : bool) returns (o : bool);
   -- make clean; make MAIN=after TRACE=trace1.txt
   ```

2. Returns true if and only if its first input has been continuously true since the last time its second input was true

   ```
   node always_since(b, a : bool) returns (o : bool);
   -- make clean; make MAIN=always_since TRACE=trace2.txt
   ```

3. Returns true if and only if its first input has been true at least once since the last time its second input was true.

   ```
   node once_since(c, a : bool) returns (o : bool);
   -- make clean; make MAIN=once_since TRACE=trace3.txt
   ```

4. Any time A has occurred in the past, either B has been continuously true, or C has occurred at least once, since the last occurrence of A

   ```
   node always_from_to(b, a, c : bool) returns (x : bool);
   -- make clean; make MAIN=always_from_to TRACE=trace4.txt
   ```

Mixing slow and fast processes

A slow process is made by undersampling with when;

A fast process is made by oversampling with current or merge.

`b`	*false*	*true*	*false*	*true*	*false*	*false*
`x`	x_0	x_1	x_2	x_3	x_4	x_5
`y`	y_0	y_1	y_2	y_3	y_4	y_5
`z = x when b`		x_1		x_3		
`k = y when not b`	y_0		y_2		y_4	y_5
`t = current z`	*nil*	x_1	x_1	x_3	x_3	x_3
`o = merge b z k`	y_0	x_1	y_2	x_3	y_4	y_5

Note that we leave "holes" in the execution trace: streams remain synchronized even when sampled.

The value of a stream is said to be present whenever it has a value at an instant and absent otherwise.

Absence is *not* represented explicitly at runtime. Instead, a special type system guarantees that an absent value is never read. A program cannot react to the absence of a value alone.

Clocks: simple example

```
node rtotal(x : int) returns (y : int);
let
  y = (0 -> pre y) + x;
tel;

node count_true(x : bool) returns (n : int);
var count : int when x;                        // <- need to specify variable "clock"
let
  count = rtotal(1 when x);
  n = current count;
tel;
```

Clocks: simple example

```
node rtotal(x : int) returns (y : int);
let
  y = (0 -> pre y) + x;
tel;

node count_true(x : bool) returns (n : int);
var count : int when x;                    // <- need to specify variable "clock"
let
  count = rtotal(1 when x);
  n = current count;
tel;
```

Sampling inputs is not the same as sampling outputs:

```
node bad_count_true(x : bool) returns (n : int);
var count : int when x;
let
  count = rtotal(1) when x;
  n = current count;
tel;
```

Sampling inputs versus sampling outputs

x	x_0	x_1	x_2	x_3	x_4	x_5	$\cdots$
x when c		x_1		x_3		x_5	$\cdots$
pre x	*nil*	x_0	x_1	x_2	x_3	x_4	$\cdots$
pre (x when c)		*nil*		x_1		x_3	$\cdots$
(pre x) when c		x_0		x_2		x_4	$\cdots$

The problem with `current`

The `current` operator is convenient. In implementation terms, it simply represents a buffer that is written less frequently than it is read.

But, its value may not be determined for several cycles, if ever.

`b`	*false*	*false*	*false*	*true*	*false*	*false*
`x`	x_0	x_1	x_2	x_3	x_4	x_5
`y`	y_0	y_1	y_2	y_3	y_4	y_5
`z = x when b`				x_3		
`t = current z`	*nil*	*nil*	*nil*	x_3	x_3	x_3

Unlike for $->$, the initialization instant may depend on inputs or arbitrary expressions. A useful static analysis, e.g., type system, is impossible.

The problem with `current`

The `current` operator is convenient. In implementation terms, it simply represents a buffer that is written less frequently than it is read.

But, its value may not be determined for several cycles, if ever.

`b`	*false*	*false*	*false*	*true*	*false*	*false*
`x`	x_0	x_1	x_2	x_3	x_4	x_5
`y`	y_0	y_1	y_2	y_3	y_4	y_5
`z = x when b`				x_3		
`t = current z`	*nil*	*nil*	*nil*	x_3	x_3	x_3

Unlike for −>, the initialization instant may depend on inputs or arbitrary expressions. A useful static analysis, e.g., type system, is impossible.

What about a runtime error?

- Less efficient: need state to track whether initialization has occurred and extra branching to test the (transitive) use of `current` values.
- Not ideal for embedded systems; better to detect as many errors as possible before a program goes into production.

Safer oversampling with merge

The merge operator requires the specification of value(s) for the complementary sampling condition.

Syntax in Heptagon: merge c (true –> et) (false –> ef)

Alternative: merge c et ef (Syntax in Scade: merge(c; et; ef))

1. Use a default (often constant) value

```
merge x (rtotal(1 when x)) 0
```

2. Program a well-initialized current

```
node current(vi : int; ck : bool; v : int :: ck) returns (b : int);
let
  b = merge ck v ((vi fby b) when not ck);
tel
```

(Sadly, not polymorphic.)

Clocks: simple example (again)

```
node current(vi : int; ck : bool; v : int :: ck) returns (b : int);
let
  b = merge ck v ((vi fby b) when not ck);
tel

node rtotal(x : int) returns (y : int);
let
  y = (0 -> pre y) + x;
tel;

node count_true(x : bool) returns (n : int);
var count : int when x;
let
  count = rtotal(1 when x);
  n = current(0, x, count);
tel;
```

- No risk of using an arbitrary value.
- Less convenient for the programmer: explicit initial value and clock.
- Initial conditions (and state variables) are often duplicated.

Alternatives to `when` and `merge`

Modern languages, like SCADE 6, provide polymorphic constructs that hide clocks, sampling, and buffering from the programmer.

1. Use a default value

```
o = (activate f every ec default ed)(e1, ..., en)
```

means

```
h = ec;
o = merge h (f((e1, ..., en) when h)) (ed when not h)
```

2. Use an initial value or the previous value

```
o = (activate f every ec initial default ed)(e1, ..., en)
```

means

```
h = ec;
o = merge h (f((e1, ..., en) when h)) ((ed -> pre o) when not h)
```

Exercise: Correct count of trues and falses

```
node tf_count (in : bool) returns (out : int);
let
  out = if in then mod_count(512) else mod_count(512); (* wrong *)
tel;
```

The Gilbreath trick

The Gilbreath shuffle (from Wikipedia):

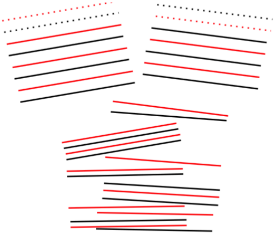
1. Deal off any number of the cards from the top of a deck onto a new pile.
2. Riffle the new pile with the remainder of the deck.

A trick based on the resulting Gilbreath permutations was formalized and verified in Coq by G. Huet. [Huet (1991): The Gilbreath trick: a case study in axiomatisation and proof development in the Coq proof assistant]

Presentation of the magic trick in G.Huet paper:

Why is this a card trick? Our boolean words are card decks, with true for red and false for black. Take an even deck x, arranged alternatively red, black, red, black, etc. Ask a spectator to cut the deck, into sub-decks u and v. Now shuffle u and v into a new deck w. When shuffling, note carefully whether u and v start with opposite colors or not. If they do, the resulting deck is composed of pairs red-black or black-red; otherwise, you get the property by first rotating the deck by one card. The trick is usually played by putting the deck behind your back after the shuffle, to perform "magic". The magic is either rotating or doing nothing. When showing the pairing property, say loudly "red black red black..." in order to confuse in the spectator's mind the weak *paired* property with the strong *alternate* one.

There is a variant. If the cut is favorable, that is if u and v are opposite, just go ahead showing the pairing, without the "magic part." If the spectator says that he understands the trick, show him the counter-example in the non-favorable case. Of course now you have to leave him puzzled, and refuse to redo the trick.



Input: two decks of alternating colours (red, black, red, black, . . .) whose bottom cards have different colours.

Output: one deck of alternating red/black pairs.

The Gilbreath trick in Scade/Lustre (thanks to J.-L. Colaço)

The property is implied by the following one on Boolean streams:

> *if* `s1` *and* `s2` *be two alternating streams starting with different values; let* `o` *be a stream built by "riffle shuffling"* `s1` *and* `s2`*, then* `o` *is such that it is the succession of pairs of different values.*

The Gilbreath trick in Scade 6

```
node Gilbreath_stream (clock c : bool) returns (prop: bool; o:bool);
var
  s1 : bool when c;
  s2 : bool when not c;
  half : bool;
let
  s1 = (false when c) -> not (pre s1);
  s2 = (true  when not c) -> not (pre s2);
  o = merge (c; s1; s2);
  half = false -> (not pre half);

  prop = true -> not (half and (o = pre o));
tel;
```

The Gilbreath trick in Lustre

```
node Gilbreath_stream (c:bool) returns (OK: bool; o:bool);
var ps1, s1 : bool;
    ps2, s2 : bool;
    half : bool;
let
  s1 = if c then not ps1 else ps1;
  ps1 = false -> pre s1;
  s2 = if not c then not ps2 else ps2;
  ps2 = true  -> pre s2;

  o = if c then s1 else s2;

  half = false -> not (pre half);

  OK = true -> not (half and (o = pre o));
tel;
```

Proved automatically using Lesar or Kind 2.

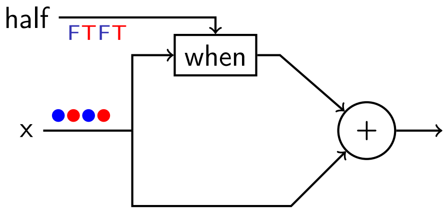
Different time scales

Synchronise slow and fast processes?

X	1	2	3	4	5	$\cdots$
$half$	T	F	T	F	T	$\cdots$
X $when$ $half$	1		3		5	$\cdots$
$X + (X$ $when$ $half)$	2		5		8	$\cdots$

```
let
  half = true -> not (pre half);
  o = x + (x when half);
tel
```

Defines the stream $\forall n \in \mathbb{N},\ o_n = x_n + x_{2n}$



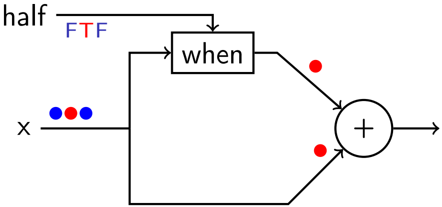
Different time scales

Synchronise slow and fast processes?

X	1	2	3	4	5	$\cdots$
$half$	T	F	T	F	T	$\cdots$
X $when$ $half$	1		3		5	$\cdots$
$X + (X$ $when$ $half)$	2		5		8	$\cdots$

```
let
  half = true -> not (pre half);
  o = x + (x when half);
tel
```

Defines the stream $\forall n \in \mathbb{N},\ o_n = x_n + x_{2n}$



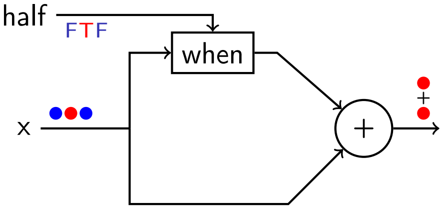
Different time scales

Synchronise slow and fast processes?

X	1	2	3	4	5	$\cdots$
half	*T*	*F*	*T*	*F*	*T*	$\cdots$
X *when half*	1		3		5	$\cdots$
$X + (X$ *when half*$)$	2		5		8	$\cdots$

```
let
  half = true -> not (pre half);
  o = x + (x when half);
tel
```

Defines the stream $\forall n \in \mathbb{N},\ o_n = x_n + x_{2n}$



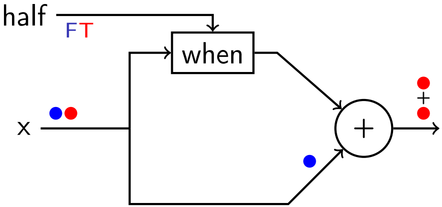
Different time scales

Synchronise slow and fast processes?

X	1	2	3	4	5	$\cdots$
$half$	T	F	T	F	T	$\cdots$
X $when$ $half$	1		3		5	$\cdots$
$X + (X$ $when$ $half)$	2		5		8	$\cdots$

```
let
  half = true -> not (pre half);
  o = x + (x when half);
tel
```

Defines the stream $\forall n \in \mathbb{N},\ o_n = x_n + x_{2n}$



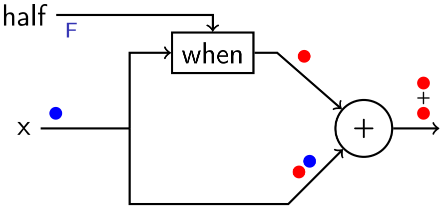
Different time scales

Synchronise slow and fast processes?

X	1	2	3	4	5	$\cdots$
$half$	T	F	T	F	T	$\cdots$
X $when$ $half$	1		3		5	$\cdots$
$X + (X$ $when$ $half)$	2		5		8	$\cdots$

```
let
  half = true -> not (pre half);
  o = x + (x when half);
tel
```

Defines the stream $\forall n \in \mathbb{N},\ o_n = x_n + x_{2n}$



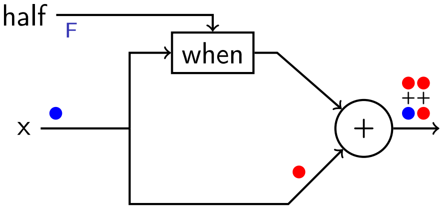
Different time scales

Synchronise slow and fast processes?

X	1	2	3	4	5	$\cdots$
$half$	T	F	T	F	T	$\cdots$
X $when$ $half$	1		3		5	$\cdots$
$X + (X$ $when$ $half)$	2		5		8	$\cdots$

```
let
  half = true -> not (pre half);
  o = x + (x when half);
tel
```

Defines the stream $\forall n \in \mathbb{N},\ o_n = x_n + x_{2n}$



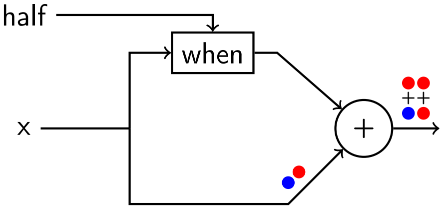
Different time scales

Synchronise slow and fast processes?

X	1	2	3	4	5	$\cdots$
$half$	T	F	T	F	T	$\cdots$
X $when$ $half$	1		3		5	$\cdots$
$X + (X$ $when$ $half)$	2		5		8	$\cdots$

```
let
  half = true -> not (pre half);
  o = x + (x when half);
tel
```

Defines the stream $\forall n \in \mathbb{N},\ o_n = x_n + x_{2n}$



Different time scales

Synchronise slow and fast processes?

X	1	2	3	4	5	$\cdots$
$half$	T	F	T	F	T	$\cdots$
X *when* $half$	1		3		5	$\cdots$
$X + (X$ *when* $half)$	2		5		8	$\cdots$

```
let
  half = true -> not (pre half);
  o = x + (x when half);
tel
```

Defines the stream $\forall n \in \mathbb{N},\ o_n = x_n + x_{2n}$

- It cannot be implemented with bounded buffers;
  (The corresponding Kahn network requires an unbounded buffer.)
- Reject such programs statically using a dedicated type system based on clock annotations. [Colaço and Pouzet (2003): Clocks as First Class Abstract Types]
- Why not just just sample? I.e., $\forall n \in \mathbb{N},\ o_n = x_n + x_{\lfloor n/2 \rfloor}$

Clock calculus

The static analysis that rejects programs that cannot be executed in finite memory is called a clock calculus. It can be expressed and implemented as a type system [Colaço and Pouzet (2003): Clocks as First Class Abstract Types].

Stream clocks s

base	base rate relative to the current node
α	abstracted base rate / undetermined clock
s on c	subsampling of s when $c = \mathit{true}$
s on not c	subsampling of s when $c = \mathit{false}$

Clock schemes σ

Abstraction for operators and nodes

$\forall\alpha,\ \forall X_1, \ldots, X_k,\ cl \times \cdots \times cl \to cl \times \cdots \times cl$

where $cl ::= s \mid (c : s)$

The type system produces judgements of the form $H \vdash e : cl$, meaning "the expression e has clock cl in the environment H".

Clock calculus (cont.)

$$+, *, \ldots : \forall\alpha,\ \alpha \times \alpha \to \alpha$$

$$\begin{aligned}
\text{pre} &: \forall\alpha,\ \alpha \to \alpha \\
\texttt{->} &: \forall\alpha,\ \alpha \to \alpha \\
\text{fby} &: \forall\alpha,\ \alpha \times \alpha \to \alpha
\end{aligned}$$

$$\begin{aligned}
\text{when} &: \forall\alpha, \forall X,\ \alpha \times (X : \alpha) \to \alpha \texttt{ on } X \\
\text{when not} &: \forall\alpha, \forall X,\ \alpha \times (X : \alpha) \to \alpha \texttt{ on not } X \\
\text{merge} &: \forall\alpha, \forall X,\ (X : \alpha) \times \alpha \texttt{ on } X \times \alpha \texttt{ on not } X \to \alpha
\end{aligned}$$

Expressions are analyzed as in any type system.

Abstraction is applied to node declarations to bind free variables.

Clock schemes are instantiated when operators or nodes are applied.

Clocks schemes in Heptagon/SCADE 6 have a single base rate (α).

Lucid synchrone [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual] allows multiple rates and higher-order functions. E.g., plumbing : $\forall\alpha\,\beta,\ \alpha \to \beta \to (\alpha \times \beta)$.

Clocks in Heptagon

The Heptagon compiler can infer clocks using type variables and unification as for standard Hindley-Milner inference (the core of OCaml).

```
-- test.ept
node sometimes_add(c : bool; x : int) returns (sum : int);
let
  sum = (0 fby sum) + (merge c x 0);
tel
```

`heptc -i test.ept` gives

```
val sometimes_add(c : bool :: .; x : int :: . on true(c))
    returns (sum : int :: .)
```

Clocks are generalized: e.g., *base* on not *c* is written `. on false(c)`.

Compare to (obligatory) clock declarations in Lustre v4: `x int when c`.

Generalization to other enumerated types

- The booleans are just an enumerated type.
- The `when` and `merge` operators, and associated clock annotations generalize naturally.

```
type mode = Rising | Falling | Stable

node counter(v : mode) returns (y : int);
var py : int;
let
  y = merge v (Rising  -> (py when Rising(v)) + 1)
              (Falling -> (py when Falling(v)) - 1)
              (Stable  -> py when Stable(v));
  py = 0 fby y;
tel
```

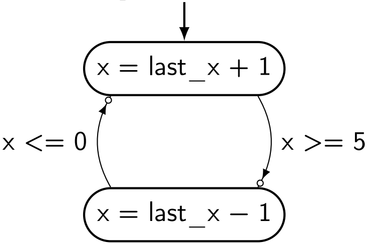
Sampling operators

- Provide a means of conditional activation.
- Programming directly with them can be tricky.
- Serve as a target for more complicated structures.
  [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]

```
node main (go : bool) returns (x : int)
  var last_x : int;
let
  last_x = 0 fby x;

  automaton
    state Up
      do x = last_x + 1
      until x >= 5 then Down

    state Down
      do x = last_x − 1
      until x <= 0 then Up
  end;
tel
```



Sampling operators

- Provide a means of conditional activation.
- Programming directly with them can be tricky.
- Serve as a target for more complicated structures.
  [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]

```
node main (go : bool) returns (x : int)
  var last_x : int;
let
  last_x = 0 fby x;

  automaton
    state Up
      do x = last_x + 1
      until x >= 5 then Down

    state Down
      do x = last_x − 1
      until x <= 0 then Up
  end;
tel
```

```
type st = St_Up | St_Down

(* ... *)

last_x = 0 fby x

x_St_Down = (last_x when St_Down(ck)) − 1
x_St_Up = (last_x when St_Up(ck)) + 1
x = merge ck (St_Down: x_St_Down)
             (St_Up: x_St_Up);

ck = St_Up fby ns
ns = ...
```

Programming with sampling [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

Rising-edge Retrigger: "Every rising edge on input *i*,
hold the output *o* true for *n* cycles."

Programming with sampling [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

Rising-edge Retrigger: "Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
let
  ...
tel
```

i	F	T	T	F	F	F	F	T	F	...
n	3	3	3	3	3	3	3	3	3	...

Programming with sampling [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

Rising-edge Retrigger: "Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge : bool;
let
  edge = i and (false fby (not i));
  ...
tel
```

i	F	T	T	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···

Programming with sampling [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

Rising-edge Retrigger: "Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge : bool; v : int;
let
  edge = i and (false fby (not i));
  v = /* count down from n whenever edge is true */
  o = v > 0;
tel
```

i	F	T	T	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···
v	0	3	2	1	0	0	0	3	2	···
o	F	T	T	T	F	F	F	T	T	···

Programming with sampling [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

Rising-edge Retrigger: "Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge : bool; v : int;
let
  edge = i and (false fby (not i));
  v = /* count down from n whenever edge is true */
  o = v > 0;
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n
        else (n fby (cpt − 1));
tel
```

i	F	T	T	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···
v	0	3	2	1	0	0	0	3	2	···
o	F	T	T	T	F	F	F	T	T	···

Programming with sampling [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

Rising-edge Retrigger: "Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = dcount((false, n) when ck);
  o = v > 0;
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n
          else (n fby (cpt − 1));
tel
```

i	F	T	T	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···
v	0	3	2	1	0	0	0	3	2	···
ck	F	T	T	T	T	F	F	T	T	···
n when ck		3	3	3	3			3	3	···
dcount(n when ck)		3	2	1	0			-1	-2	···
o	F	T	T	T	F	F	F	T	T	···

Programming with sampling [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

Rising-edge Retrigger: "Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck
      (true −> dcount((false, n) when ck))
      (false −> 0);
  o = v > 0;
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n
        else (n fby (cpt − 1));
tel
```

i	F	T	T	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···
v	0	3	2	1	0	0	0	3	2	···
ck	F	T	T	T	T	F	F	T	T	···
n when ck		3	3	3	3			3	3	···
dcount(n when ck)		3	2	1	0			-1	-2	···
o	F	T	T	T	F	F	F	T	T	···

Programming with sampling [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual]

Rising-edge Retrigger: "Every rising edge on input *i*, hold the output *o* true for *n* cycles."

```
node rising_edge_retrigger(i : bool; n : int)
returns (o : bool)
var edge, ck : bool; v : int;
let
  edge = i and (false fby (not i));
  ck = edge or (false fby o);
  v = merge ck
        (true -> dcount((edge, n) when ck))
        (false -> 0);
  o = v > 0;
tel
```

```
node dcount(res : bool; n : int)
returns (cpt : int)
let
  cpt = if res then n
          else (n fby (cpt - 1));
tel
```

i	F	T	T	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	3	···
edge	F	T	F	F	F	F	F	T	F	···
v	0	3	2	1	0	0	0	3	2	···
ck	F	T	T	T	T	F	F	T	T	···
n when ck		3	3	3	3			3	3	···
dcount(n when ck)		3	2	1	0			3	2	···
o	F	T	T	T	F	F	F	T	T	···

Dataflow programming languages

- Kahn Networks [Kahn (1974): The Semantics of a Simple Language for Parallel Programming]
- Lucid [Wadge and Ashcroft (1985): LUCID, the dataflow programming language]

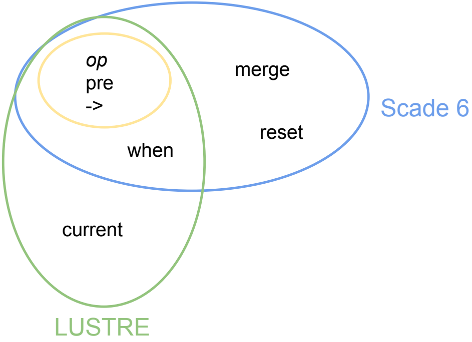
Dataflow programming languages

- Kahn Networks [Kahn (1974): The Semantics of a Simple Language for Parallel Programming]
- Lucid [Wadge and Ashcroft (1985): LUCID, the dataflow programming language]
- Lustre [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]
  - Clock calculus
  - Deterministic, bounded memory, bounded execution time
- Lucid Synchrone [Caspi and Pouzet (1995): A Functional Extension to Lustre]
  - Higher-order dataflow, Hierarchical automata, Signals
- Scade 6 [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]
  - Industrial, extended version of Lustre
  - Used in critical systems (DO-178B certified)
- Signal [Le Guernic, Gautier, Le Borgne, and Le Maire (1991): Programming Real-Time Applications with Signal]

Dataflow programming languages

- Kahn Networks [Kahn (1974): The Semantics of a Simple Language for Parallel Programming]
- Lucid [Wadge and Ashcroft (1985): LUCID, the dataflow programming language]
- Lustre [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]
  - Clock calculus
  - Deterministic, bounded memory, bounded execution time
- Lucid Synchrone [Caspi and Pouzet (1995): A Functional Extension to Lustre]
  - Higher-order dataflow, Hierarchical automata, Signals
- Scade 6 [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]
  - Industrial, extended version of Lustre
  - Used in critical systems (DO-178B certified)
- Signal [Le Guernic, Gautier, Le Borgne, and Le Maire (1991): Programming Real-Time Applications with Signal]
- Ptolemy II [(2014): System Design, Modeling, and Simulation using Ptolemy II]
- MathWorks Simulink (and Stateflow) `https://www.mathworks.com/products/simulink/`
- National Instruments LabView `https://www.ni.com/en-us/shop/labview.html`

SCADE 6 and LUSTRE kernels



[Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]

Further reading

- [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]
- [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]

References I

- Biernacki, D., J.-L. Colaço, G. Hamon, and M. Pouzet (June 2008). "Clock-directed modular code generation for synchronous data-flow languages". In: *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. Tucson, AZ, USA: ACM Press, pp. 121–130.
- Caspi, P., D. Pilaud, N. Halbwachs, and J. A. Plaice (Jan. 1987). "LUSTRE: A declarative language for programming synchronous systems". In: *Proc. 14th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1987)*. Munich, Germany: ACM Press, pp. 178–188.
- Caspi, P. and M. Pouzet (May 1995). "A Functional Extension to Lustre". In: *International Symposium on Languages for Intentional Programming*. Ed. by M. A. Orgun and E. A. Ashcroft. Sydney, Australia: World Scientific.
- Colaço, J.-L., B. Pagano, and M. Pouzet (Sept. 2005). "A Conservative Extension of Synchronous Data-flow with State Machines". In: *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*. Ed. by W. Wolf. Jersey City, USA: ACM Press, pp. 173–182.

References II

- — (Sept. 2017). "Scade 6: A Formal Language for Embedded Critical Software Development". In: *Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE 2017)*. Nice, France: IEEE Computer Society, pp. 4–15.
- Colaço, J.-L. and M. Pouzet (Oct. 2003). "Clocks as First Class Abstract Types". In: *Proc. 3rd Int. Workshop on Embedded Software (EMSOFT 2003)*. Ed. by R. Alur and I. Lee. Vol. 2855. LNCS. Philadelphia, PA, USA: Springer, pp. 134–155.
- — (Aug. 2004). "Type-based initialization analysis of a synchronous dataflow language". In: *Int. J. Software Tools for Technology Transfer* 6.3, pp. 245–255.
- Dvorak (ed.), D. L. (Mar. 2009). *NASA Study on Flight Software Complexity*. Final Report. NASA Office of Chief Engineer.
- Halbwachs, N., F. Lagnier, and P. Raymond (June 1993). "Synchronous observers and the verification of reactive systems". In: *Proc. 3rd Int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*. Ed. by M. Nivat, C. Rattray, T. Rus, and G. Scollo. Twente: Workshops in Computing, Springer Verlag.
- Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud (Sept. 1991). "The synchronous dataflow programming language LUSTRE". In: *Proc. IEEE* 79.9, pp. 1305–1320.

References III

- Halbwachs, N., J.-C. Fernandez, and A. Bouajjani (Apr. 1993). “An executable temporal logic to express safety properties and its connection with the language Lustre”. In: *Proc. 6th Int. Symp. Lucid and Intensional Programming (ISLIP’93)*. Quebec, Canada.
- Halbwachs, N., F. Lagnier, and C. Ratel (Sept. 1992). “Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE”. In: *IEEE Trans. Software Engineering* 18.9, pp. 785–793.
- Huet, G. (Sept. 1991). *The Gilbreath trick: a case study in axiomatisation and proof development in the Coq proof assistant*. Rapport de Recherche RR-1511. Rocquencourt, France: Inria.
- Jahier, E., P. Raymond, and N. Halbwachs (May 2019). *The Lustre V6 Reference Manual*. Verimag. Grenoble.
- Kahn, G. (Aug. 1974). “The Semantics of a Simple Language for Parallel Programming”. In: *Proc. 6th Int. Federation for Information Processing (IFIP) Congress 1974*. Ed. by J. L. Rosenfeld. Stockholm, Sweden: North-Holland, pp. 471–475.

References IV

- Le Guernic, P., T. Gautier, M. Le Borgne, and C. Le Maire (Sept. 1991). "Programming Real-Time Applications with Signal". In: *Proc. IEEE* 79.9, pp. 1321–1336.
- Lublinerman, R., C. Szegedy, and S. Tripakis (Jan. 2009). "Modular Code Generation from Synchronous Block Diagrams: Modularity vs. Code Size". In: *Proc. 36th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2009)*. Savannah, GA, USA: ACM Press, pp. 78–89.
- Plaice, J. A. (1988). "Sémantique et compilation de LUSTRE, un langage déclaratif synchrone". PhD thesis. Grenoble INP.
- Pouzet, M. (Apr. 2006). *Lucid Synchrone, v. 3. Tutorial and reference manual*. Université Paris-Sud.
- Pouzet, M. and P. Raymond (Oct. 2009). "Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation". In: *Proc. 9th ACM Int. Conf. on Embedded Software (EMSOFT 2009)*. Grenoble, France: ACM Press, pp. 215–224.
- Ptolemaeus, C., ed. (2014). *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org.

References V

- Raymond, P. (1991). "Compilation efficace d'un langage déclaratif synchrone: le générateur de code Lustre-V3". PhD thesis. Grenoble INP.
- — (July 1996). "Recognizing regular expressions by means of dataflow networks". In: *Proc. 23rd Int. Colloq. on Automata, Languages and Programming*. Ed. by F. Meyer auf der Heide and B. Monien. LNCS 1099. Paderborn, Germany: Springer, pp. 336–347.
- Raymond, P., Y. Roux, and E. Jahier (2008). "Lutin: A Language for Specifying and Executing Reactive Scenarios". In: *EURASIP Journal of Embedded Systems*.
- The Economist (June 2012). "Open-source medical devices: When code can kill or cure". In: *The Economist: Technology Quarterly*.
- Wadge, W. W. and E. A. Ashcroft (1985). *LUCID, the dataflow programming language*. Academic Press Professional, Inc.