Programmtion synchrone

Marc.Pouzet@ens.fr [1]

Décembre 2019

[1] Notes de cours préparées avec Timothy Bourke (INRIA)

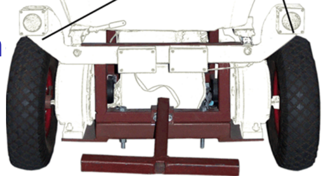
Today

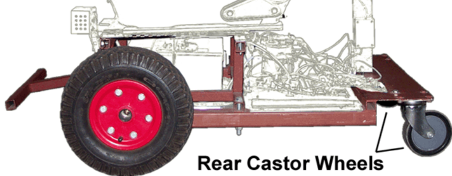
- Motivation
- Some historical context
- Programming in Lustre and Scade 6
- Programming in Zélus

Wheelchair: An old, simple, but concrete example

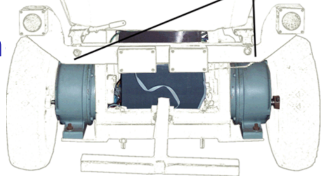
- The UOW ‘robotic’ wheelchair
- Goal: low-cost mobility assistance
- Target of engineering student projects

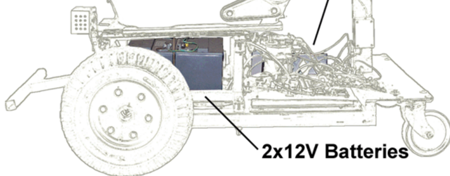
Wh



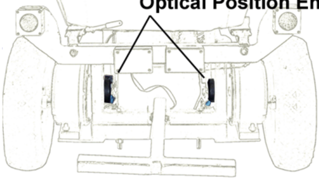


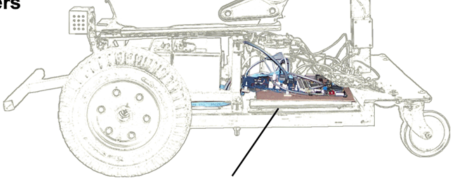
Wh



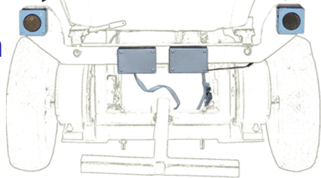


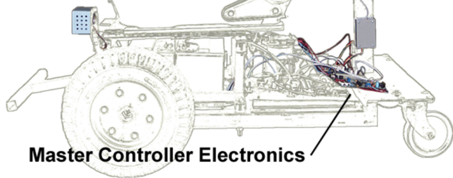
Wh



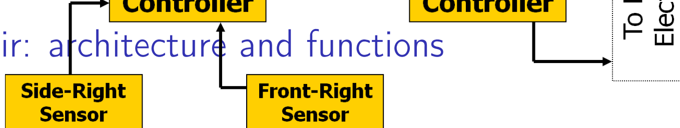


Wh

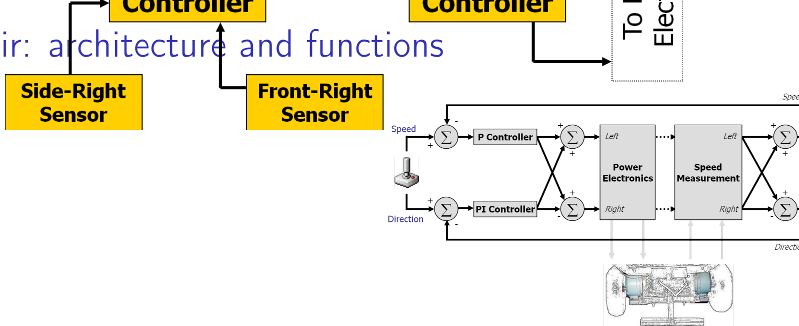




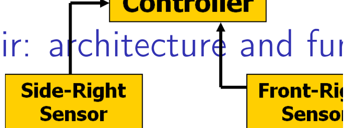
Wheelchair: architecture and functions

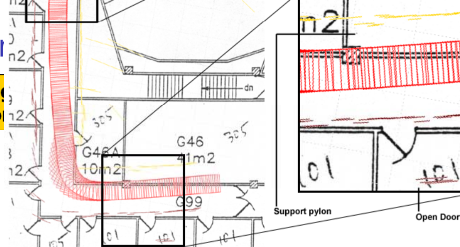


Wheelchair: architecture and functions



Wheelchair: architecture and fu





Wheelchair: drive control feedback loop

Speed

Direction

P Controller

PI Controller

Power Electronics

Left

Right

Speed Measurement

Left

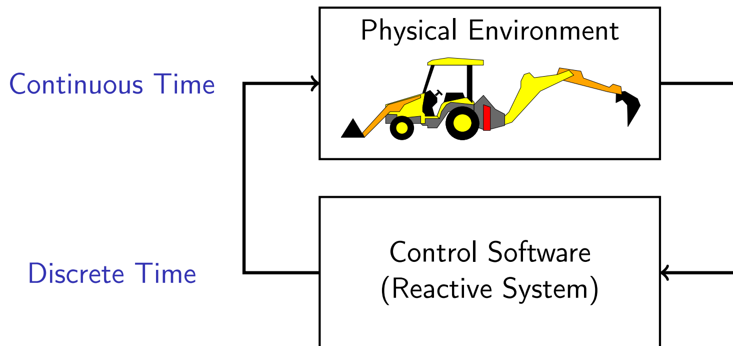
Right

Speed

Direction

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

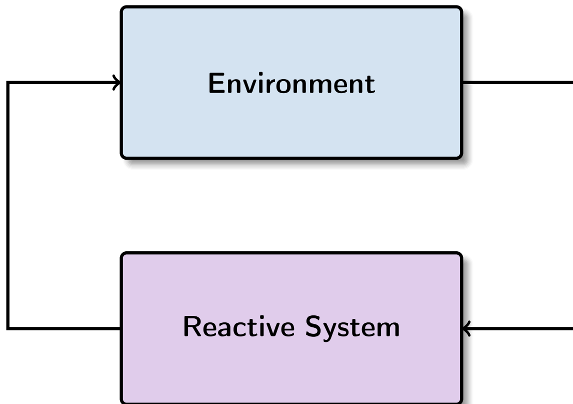
Embedded Real-Time Software



- 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

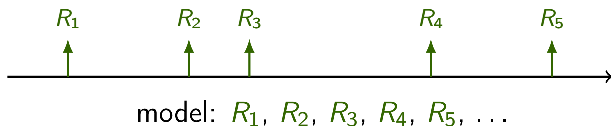
[Dvorak (ed.) (2009): NASA Study on Flight Software Complexity] [The Economist (2012): Open-source medical devices: When code can kill or cure]

Program and control a reactive system?



Cyclic execution

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



- 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.
- How to compose such systems?

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?

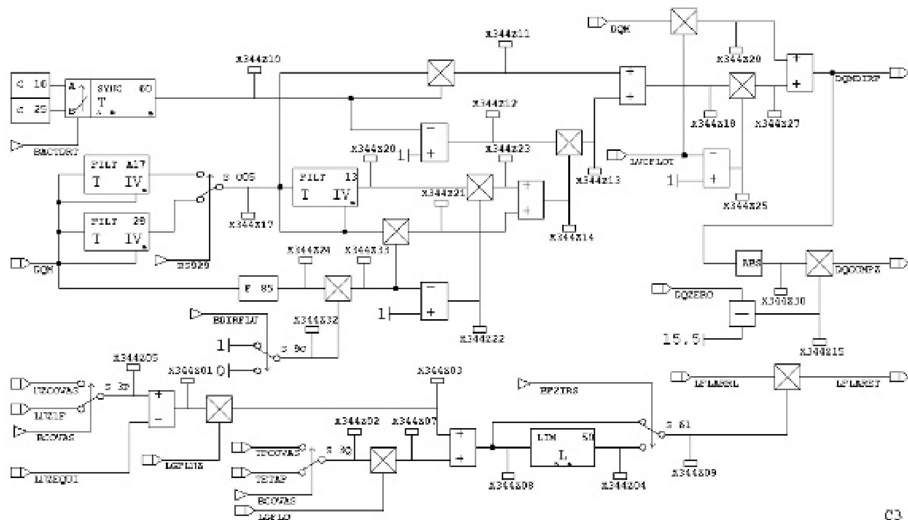
- What is the programmer's model of time?
- How are timing details expressed, implemented, validated?

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

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

What is the specification?

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



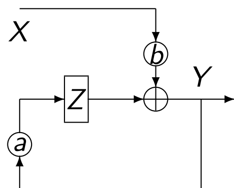
They were very precise drawings...

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

Stream equations, z-transform, etc.

Example: a linear filter

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



...but not executable. Write code and convince oneself that it is correct?

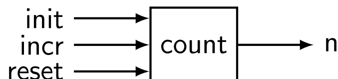
How to make those mathematics executable?

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

[Caspi, Pilaud, Halbwachs, and J. 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;
```

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"

Well... is it real-time?

Reason in worst case: verify that the generated code produces its output before the arrival of the next input.

Example: 1-bit adder

```
node full_add(a, b, c:bool) returns (s, co:bool);
let
  s = (a xor b) xor c;
  co = (a and b) or (b and c) or (a and c);
tel;
```

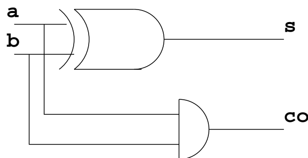
or:

```
node full_add(a, b, c:bool) returns (s, co:bool);
let
  s = (a xor b) xor c;
  co = if a then b or c else b and c;
tel;
```

Full Adder

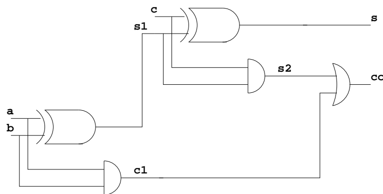
Compose two "half adder"

```
node half_add(a, b : bool)
returns (s, co : bool);
let s = a xor b;
    co = a and b;
tel;
```



Instantiate it twice:

```
node full_add_h(a, b, c : bool)
returns (s, co : bool);
var s1, c1, c2 : bool;
let
  (s1, c1) = half_add(a,b);
  (s, c2) = half_add(c, s1);
  co = c1 or c2;
tel;
```



Verify properties

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

$$\forall a, b, c : \texttt{bool}.\ \texttt{full_add}(a, b, c) = \texttt{full_add_h}(a, b, c)$$

Write the following program and prove that it returns `true` at every instant.

```
-- file fulladder.lus
node equivalence(a, b, c : bool) returns (ok : bool);
let
  -- TODO: write equivalence property
  ok = true;
tel;
```

Then, use the model-checking tool `lesar`:

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

TRUE PROPERTY
```

Exercise: simulation/verification of a stateless program

- Download and uncompress the Verimag Lustre v4 toolset:
  `http://www-verimag.imag.fr/The-Lustre-Toolbox.html?lang=en`
- Set up the environment variables:
  ```
  export LUSTRE_INSTALL=$(pwd)
  source $LUSTRE_INSTALL/setenv.sh
  ```
- Download
  `https://www.di.ens.fr/~pouzet/cours/mpri/cours1/fulladder.lus`
- Simulate nodes with `luciole`: `luciole fulladder.lus full_add`
- Write the `equivalence` property and verify it with `lesar`:
  ```
  -- file fulladder.lus
  node equivalence(a, b, c : bool) returns (ok : bool);
  let
    -- TODO: write equivalence property
    ok = true;
  tel;
  ```
  ```
  lesar fulladder.lus equivalence
  ```

Verify properties

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

$$\forall a, b, c : \texttt{bool}.\ \texttt{full_add}(a, b, c) = \texttt{full_add_h}(a, b, c)$$

Write the following program and prove that it returns `true` at every instant.

```
-- file fulladder.lus
node equivalence(a,b,c:bool) returns (ok:bool);
  var o1, c1, o2, c2: bool;
  let
    (o1, c1) = full_add(a,b,c);
    (o2, c2) = full_add_h(a,b,c);
    ok = (o1 = o2) and (c1 = c2);
  tel;
```

Then, use the model-checking tool `lesar`:

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

TRUE PROPERTY
```

The Unit Delay

One can refer to the value of an input at the "previous" step.

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	$\cdots$

- The stream $(S_n)_{n \in \mathbb{N}}$ with $S_0 = X_0$ and $S_n = S_{n-1} + X_n$, for $n > 0$ is written:

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

- Introducing intermediate equations does not change the meaning of programs:

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

- The `fby` (*followed by*) operator combines `pre` and `->`:

  `X fby Y` defines the same stream as `X -> (pre Y)`

Installing the Heptagon Lustre compiler

- Heptagon web site: `http://heptagon.gforge.inria.fr`
- Download the source code package and uncompress it.
- Install prerequisites with opam

```
opam install ocamlfind menhir ocamlgraph camlp4 lablgtk
```

- Build and install Heptagon:
  - » `./configure`
  - » `make -j`
  - » `make install`

- There is a manual and an `examples` subdirectory.

Exercise: simple filtering

- Consider the input signal:

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

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

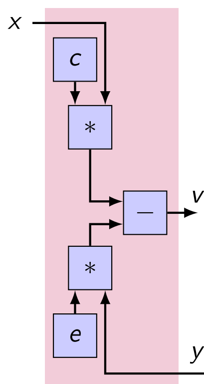
$$y(n) = \frac{1}{3}\left(\sum_{k=0}^{2} x(n-k)\right)$$

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

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

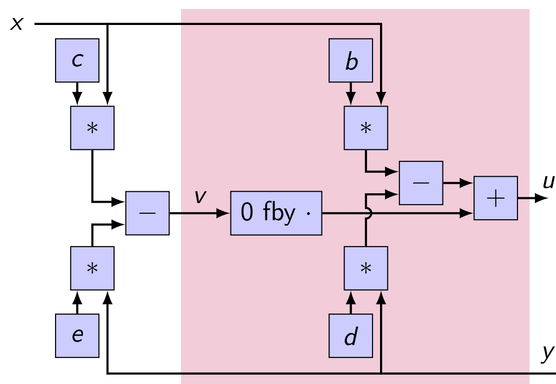
- Both are basic FIR (Finite Impulse Response) filters.

Dataflow programming: composition and syntax




```
v = c * x − e * y;
```

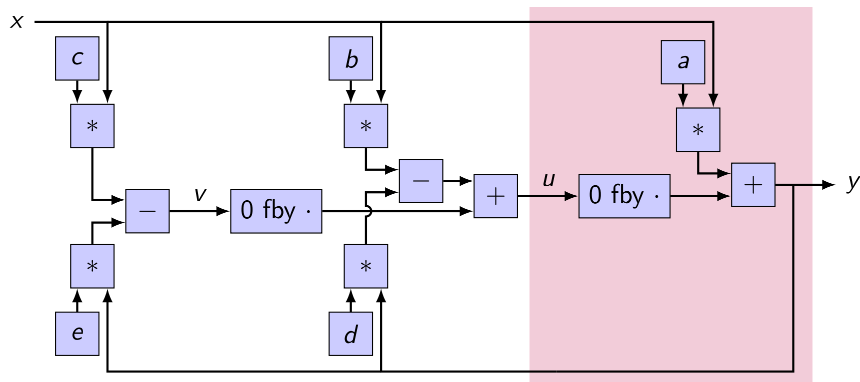
Dataflow programming: composition and syntax



$$\mathtt{u} = \mathtt{b} * \mathtt{x} - \mathtt{d} * \mathtt{y}\ + (\mathtt{0.0\ fby\ v});$$

$$\mathtt{v} = \mathtt{c} * \mathtt{x} - \mathtt{e} * \mathtt{y};$$

Dataflow programming: composition and syntax

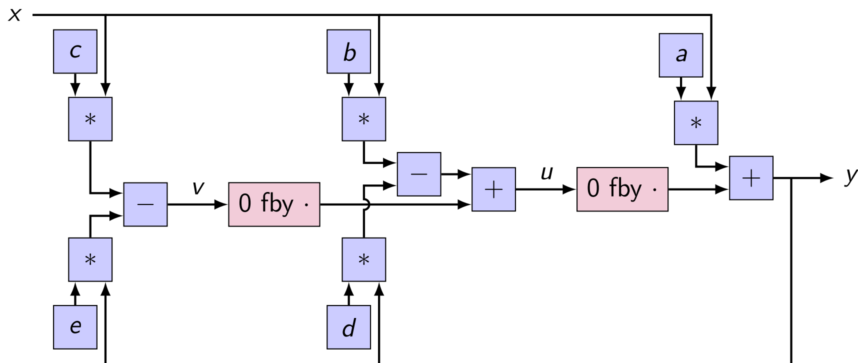


$$y = a * x + (0.0 \text{ fby } u);$$

$$u = b * x - d * y + (0.0 \text{ fby } v);$$

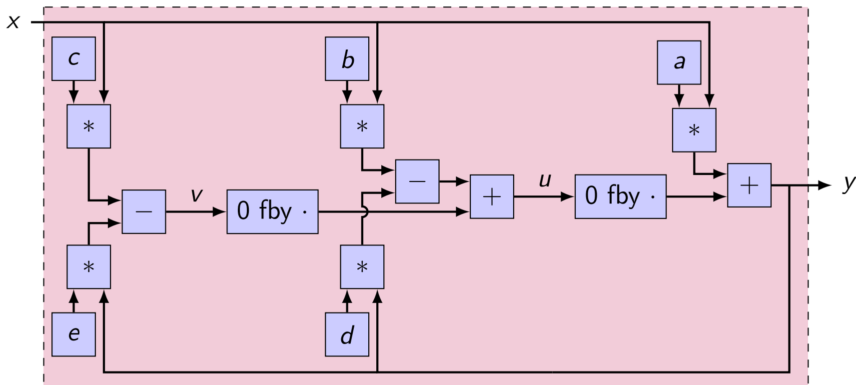
$$v = c * x - e * y;$$

Dataflow programming: composition and syntax




```
y = a * x  + (0.0 fby u);
u = b * x − d * y  + (0.0 fby v);
v = c * x − e * y;
```

Dataflow programming: composition and syntax




```
node iir_filter_2(x : real) returns (y : real);
let
        y = a * x  + (0.0 fby u);
        u = b * x − d * y  + (0.0 fby v);
        v = c * x − e * y;
tel;
```

Analyse causality between signals

Some programs have zero solution (*deadlock*) or too many (*non determinism*)

In Lustre/Signal

- `x = y + 1 and y = x + 2`
- `y = x and x = y`

In Esterel

- `present S else emit S`
- `present S1 then emit S2 || present S2 else emit S1`

Discovery

A "good" notion of causality is that of electricity.

If we "cable" the synchronous program, do the output become stable?

Coincide with what can be proved in constructive logic.

Modular causality analysis

Should these programs be accepted?

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

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

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

node main2(w : int)
returns (z : int);
let
  z = 0 -> 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
```

It depends on the compilation schema.

- Inlining and (minimize) automaton generation (Lustre v4)
  [J. A. 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]

Example: the heat controller (N. Halwachs, CdF 2010)

Model of the environment

- u is the command: *true* means heater on; *false* means heater off.
- α, β, c are static parameters; *ext* is the external temperature.
- The rate of change $temp'$ of the temperature $temp$ evolves according to:

$$temp' = \begin{cases} \alpha(c - temp) & \text{if } u = \texttt{true} \\ \beta(ext - temp) & \text{if } u = \texttt{false} \end{cases}$$

We discretize with a step of size h

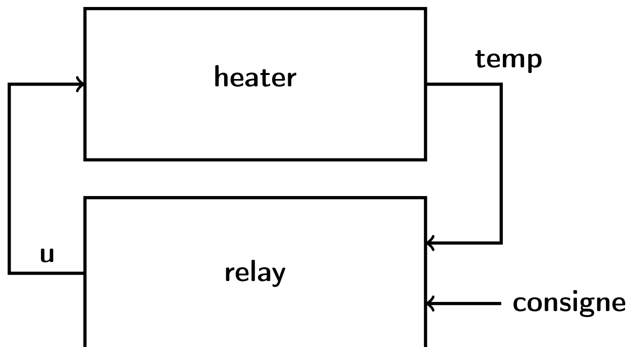
$temp'$ is approximated by the difference $(temp_{n+1} - temp_n)/h$

Relay controller

$$u_{n>0} = \begin{cases} true & \text{if } temp_n < bas \\ false & \text{if } temp_n > haut \\ u_{n-1} & \text{otherwise} \end{cases}$$

$$u_0 = false$$

Exercise: program the heat controller




```
-- heat.lus
node main(consigne : real) returns (u : bool; temp : real);
```

```
-- Lustre v4 simulator
luciole heat.lus main
```

Counting Beacons (Example due to P. Raymond)

Counting becons and seconds to decide whether a train is on time.

Use an hysteresis with a low and high threshold to reduce oscillations.

```
node beacon(sec, bea: bool) returns (ontime, late, early: bool);
var diff, pdiff: int; pontime: bool;
let
  pdiff = 0 -> pre diff;
  diff = pdiff + (if bea then 1 else 0) +
         (if sec then -1 else 0);
  early = pontime and (diff > 3) or
          (false -> pre early) and (diff > 1);
  late = pontime and (diff < -3) or
         (false -> pre late) and (diff < -1);
  ontime = not (early or late);
  pontime = true -> pre ontime;
tel;
```

Two types of properties

Safety property

"Something bad never happens", i.e., a property is invariant and true in any accessible state. E.g.:

- "The train is never both early and late", it is either on time, late or early;
- "The train never passes immediately from late to early"
- "It is impossible to stay late only a single instant".

Liveness property

"Something good eventually happens.", i.e., any execution will reach a state verifying the property.
E.g., "If the train stops, it will eventually be late."

Remark:

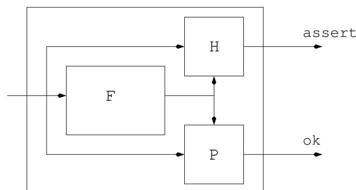
"If the train is on time and stops for ten seconds, it will be eventually late" is a safety property.
Safety properties are critical ones in practice.

Synchronous observers

Express (safety) properties in the same language as the program and compose them in synchronous parallel for verification.

1. System to verify: $y = F(x)$;
2. Hypothesis on the environment: $\texttt{assert}(H(x, y))$;
3. Property relating inputs x and outputs y: $ok = P(x, y)$.

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



If *assert* is always true, then so is *ok*: $\texttt{always}(assert) \Rightarrow \texttt{always}(ok)$.

Safety temporal properties are expressed as observers.

[Halbwachs, Lagnier, and Raymond (1993): Synchronous observers and the verification of reactive systems] [Halbwachs, Fernandez, and Bouajjani (1993): An executable temporal logic to express safety properties and its connection with the language Lustre].

Model checking

- Lesar: based on BDDs

  [Halbwachs, Lagnier, and Ratel (1992): Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE]

- Kind 2: based SMT/k-induction/IC3

  [Champion, Mebsout, Sticksel, and Tinelli (2016): The Kind 2 Model Checker]

- DV of (Ansys) Scade based on Prover SAT/k-induction

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

Clocks: mixing slow and fast processes

A slow process is made by under-sampling inputs;
A fast one by oversampling inputs.

The operators `when` and `current`

`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

Clocks: simple example

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

node count_bananas(banana : bool) returns (n : int);
var count : int when banana;
let
  count = cumulative_sum(1 when banana);
  n = current count;
tel;
```

Clocks: simple example

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

node count_bananas(banana : bool) returns (n : int);
var count : int when banana;
let
  count = cumulative_sum(1 when banana);
  n = current count;
tel;
```

Sampling the inputs is not the same as sampling the outputs:

```
node bad_count_bananas(banana : bool) returns (n : int);
var count : int when banana;
let
  count = cumulative_sum(1) when banana;
  n = current count;
tel;
```

Exercise: periodic relay

Modify the heater example to execute the relay process with a period of $4h$, where h is the model step size.

Scade 6: merge

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

Scade 6: merge

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	···
sec	F	F	F	T	F	T	T	F	···

Scade 6: merge

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯

Scade 6: merge

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯

Scade 6: merge

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	...
sec	F	F	F	T	F	T	T	F	...
r	0	1	3	4	6	9	9	12	...
(c_1)	0	0	1	3	4	6	9	9	...
r when sec				4		9	9		...
t				1		2	3		...
(c_2)				0		1	2		...

Scade 6: merge

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	···
sec	F	F	F	T	F	T	T	F	···
r	0	1	3	4	6	9	9	12	···
(c_1)	0	0	1	3	4	6	9	9	···
r when sec				4		9	9		···
t				1		2	3		···
(c_2)				0		1	2		···
0 fby v	0	0	0	0	4	4	4	3	···
(0 fby v) when not sec	0	0	0		4			3	···

Scade 6: merge

```
node avgvelocity(delta: int; sec: bool) returns (r, v: int)
  var t : int;
let
  r = count(0, delta, false);
  t = count((1, 1, false) when sec);
  v = merge sec ((r when sec) / t) ((0 fby v) when not sec);
tel
```

delta	0	1	2	1	2	3	0	3	⋯
sec	F	F	F	T	F	T	T	F	⋯
r	0	1	3	4	6	9	9	12	⋯
(c_1)	0	0	1	3	4	6	9	9	⋯
r when sec				4		9	9		⋯
t				1		2	3		⋯
(c_2)				0		1	2		⋯
0 fby v	0	0	0	0	4	4	4	3	⋯
(0 fby v) when not sec	0	0	0		4			3	⋯
v	0	0	0	4	4	4	3	3	⋯

A well-initialized current

Scade 6 syntax:

```
node current(v_i : int; v : int when ck; ck : bool) returns (c : int)
var pc : int;
let
  c = merge(ck; v; pc whenot ck);
  pc = v_i -> pre c;
tel
```

A well-initialized current

Scade 6 syntax:

```
node current(v_i : int; v : int when ck; ck : bool) returns (c : int)
var pc : int;
let
  c = merge(ck; v; pc whenot ck);
  pc = v_i -> pre c;
tel
```

Heptagon syntax (`http://heptagon.gforge.inria.fr`):

```
node current(v_i : int; v : int :: ck; ck : bool) returns (c : int)
var pc : int;
let
  c = merge ck v (pc whenot ck);
  pc = v_i -> pre c;
tel
```

The Gilbreath trick

The Gilbreath shuffle (from Wikipedia):

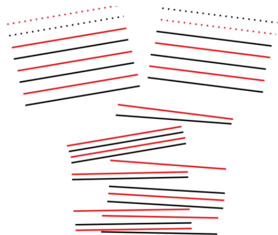
1. Deal off any number of the cards from the top of a deck onto a new pile.
2. Riffle the new pile with the remainder of the deck.

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 $\forall n \in \mathbb{N}.\ o_n = x_n + x_{2n}$

- It cannot be implemented with bounded buffers;
  (The corresponding Kahn network requires an unbounded buffer.)
- Reject such programs statically using a dedicated type system.

Sampling and merging: what for?

- Provide a means of conditional activation.
- Programming directly with them can be tricky.
- Serve as a target for more complicated structures.

```
node example(input : int32; clk : bool)
returns (o : int32 last = 0)
let
  activate if clk
  then o = f input
  else
  returns o;
tel
```

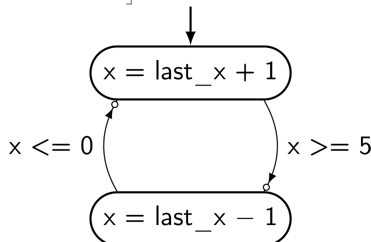
```
node example(input : int32; clk : bool)
returns (o : int32)
let
  o = merge(clk; f (input when clk);
            last_o when not clk);
  last_o = 0 -> (pre o);
tel
```

```
let node cond_act f clk default input =
  let rec o =
    merge clk (run f (input when clk))
              ((default fby o) whenot clk) in
  o
```

Sampling and merging: what for?

- 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 and merging: what for?

- 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 = ...
```

Modular Reset

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

Passing extra inputs everywhere to enable resets is tedious and inefficient.

Modular Reset

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

Passing extra inputs everywhere to enable resets is tedious and inefficient.

Scade 6 has a modular reset construction [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs].

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

(* Scade 6: expression-based *)
node main1(r : bool)
returns (n : int);
let
  n = (restart COUNT every r)(0, 1);
tel
```

```
(* Heptagon: block-based *)
node main2(r : bool)
returns (n : int);
let
  reset
    n = COUNT (0, 1);
  every r;
tel
```

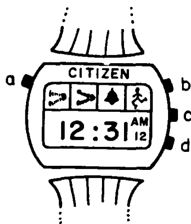


Fig. 7.

- Harel's Statecharts [Harel (1987): Statecharts: A Visual Formalism for Complex Systems]

Citizen quartz multi-alarm
main
displays
regular
update
wait
date
beep-test
time
stopwatch
out
chime
alarm 2
update 2
alarm 1
update 1
alarms-beep
alarm 1 beeps
alarm 2 beeps
both beep
T hits T1 (P1)
any button pressed
30 sec in alarms-beep
T hits T2 (P2)
T hits T1 (P)
2 min in out (no changes)
2 min in update
2 min in date
alarm 1 - status
alarm 2 - status
chime - status
light
power
enabled
disabled
quiet
beep
T is whole hour
2 sec in beep
battery inserted
battery removed /clh(main*)
dead
battery weakens
weak battery dies /clh(main*)
blink

Fig. 31

Statecharts

Original paper full of great ideas, but what do the diagrams mean?
How should they be executed?

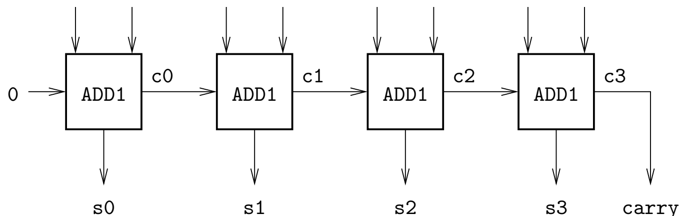
Response of synchronous languages:

- Argos and Mode Automata
  [Maraninchi and Rémond (2001): Argos: an automaton-based synchronous language] [Maraninchi and Rémond (2003): Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems]
- Esterel
  [Berry (2000): The Esterel v5 Language Primer]
  [Berry (1989): Programming a Digital Watch in Esterel v3]
- SyncCharts $\rightarrow$ Safe State Machines
  [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]
- Scade 6 State Machines
  [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]

Arrays: Lustre v4 (1/2)

- Example from §3 [Halbwachs and Raymond (2007): A Tutorial of Lustre]
- Without arrays:

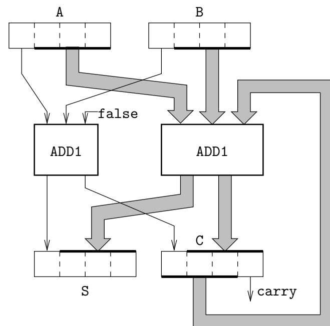
```
node first_add4 (a0,a1,a2,a3: bool; b0,b1,b2,b3: bool)
returns (s0,s1,s2,s3:bool; carry: bool);
var c0,c1,c2,c3: bool;
let
  (s0,c0) = full_add(a0,b0,false);
  (s1,c1) = full_add(a1,b1,c0);
  (s2,c2) = full_add(a2,b2,c1);
  (s3,c3) = full_add(a3,b3,c2);
  carry = c3;
tel
```



Arrays: Lustre v4 (2/2)

Same example with arrays [Halbwachs and Raymond (2007): A Tutorial of Lustre]:

```
node add4 (A, B: bool^4) returns (S: bool^4; carry: bool);
var C: bool^4;
let
  (S[0], C[0]) =  full_add(A[0], B[0], false);
  (S[1..3], C[1..3]) = full_add(A[1..3], B[1..3], C[0..2]);
  carry = C[3];
tel
```



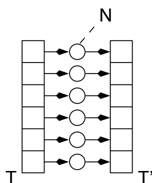
- Recursive expansion of expressions during compilation.
- More suitable for hardware than for software.

Arrays: Heptagon (1/2)

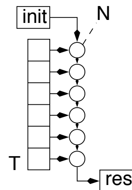
- Functional arrays with iterators [Developers (2017): Heptagon/BZR manual]
- Semi-linear typing to eliminate unnecessary copies (use `-O` [Gérard, Guatto, Pasteur, and Pouzet (2012): A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler]

Access with a constant index	`t[4]`
Access with a dynamic index	`t.[x] default v`
(truncated to 0 or $n-1$)	`t.[>x<]`
Modify an element	`[ t with [x] = v ]`
Create by replication	`x^n`
Create explicitly	`[1, x, 3, y, 5]`
Extract a slice	`t[n..m]`
Concatenation	`t1 @ t2`

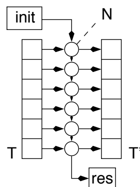
Arrays: Heptagon (2/2)



(a) map



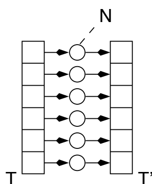
(b) fold



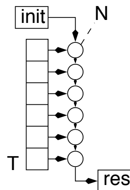
(c) mapfold

```
fun f(x, y : int) returns (r : int);
let
  r = x + 10 * y;
tel

node go<<n : int>>(a, b : int^n) returns (s : int^n);
let
  s = map<<n>> f (a, b);
tel
```

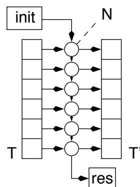
Arrays: Heptagon (2/2)



(a) map



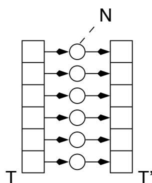
(b) fold



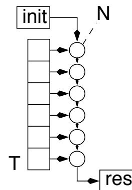
(c) mapfold

```
fun f_i(x, y, i : int) returns (r1, r2 : int);
let
  r1 = x + 10 * y;
  r2 = i;
tel

node go_i<<n : int>>(a, b : int^n) returns (s1, s2 : int^n);
let
  (s1, s2) = mapi<<n>> f_i (a, b);
tel
```

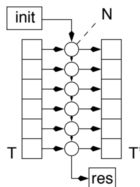
Arrays: Heptagon (2/2)



(a) map



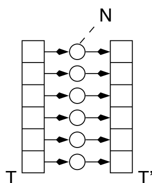
(b) fold



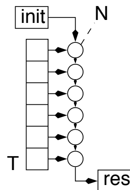
(c) mapfold

```
fun f(x, y : int) returns (r : int);
let
  r = x + 10 * y;
tel

node go_f<<n : int>>(a : int^n) returns (s : int);
let
  s = fold<<n>> f (a, 0);
tel
```

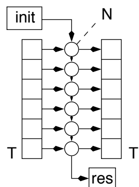
Arrays: Heptagon (2/2)



(a) map



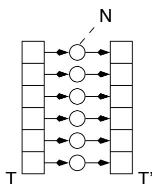
(b) fold



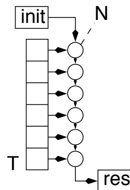
(c) mapfold

```
fun g(x, y, i, acc : int) returns (acc' : int);
let
  acc' = if (i % 2 = 0) then (x + y + acc) else acc;
tel

node go_fi<<n : int>>(a, b : int^n) returns (s : int);
let
  s = foldi<<n>> g (a, b, 0);
tel
```

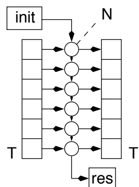
Arrays: Heptagon (2/2)



(a) map



(b) fold



(c) mapfold

```
node full_add(a, b, c : bool) returns (s, co : bool);
let
  s = (a <> b) <> c;
  co = if a then b or c else b and c;
tel;

node add4 (a, b: bool^4) returns (s: bool^4; carry: bool);
let
  (s, carry) = mapfold<<4>> full_add (a, b, false);
tel
```

Exercise: *n*-place FIFOs

- Program an *n*-place FIFO using Heptagon's arrays.

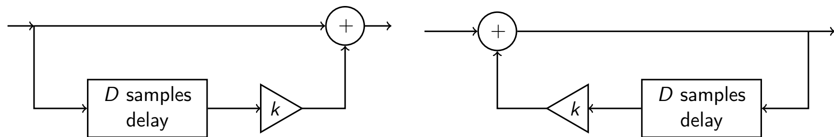
```
node fifo<<n, y0:int>>(x : int) returns (y : int);
```

- Use it to calculate a sliding average.

```
node sliding_average<<n : int>>(x : int)
returns (avg : int);
```

- Use it to generate echo effects on audio signals. (e.g., see https://sound.eti.pg.gda.pl/student/eim/synteza/adamx/eindex.html)

```
node main(ic1, ic2 : float) returns (oc1, oc2 : float);
```



Exercise: FIFOs and clocks

- The '*n*-in-window' specification.

```
node ninw<<w, n:int>>(new_x : bool; x : int when new_x)
returns (valid : bool; r : int^n when valid);
```

- Receive inputs on x from time-to-time. If n such elements have been received in the last w ticks then return them in any order.

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

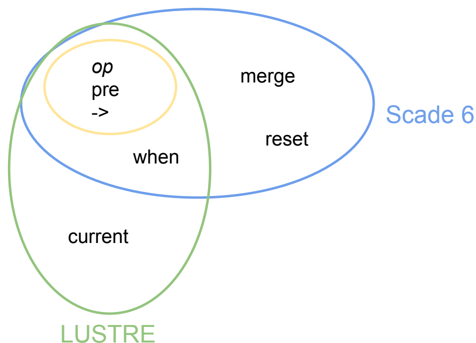
- Kahn Networks [Kahn (1974): The Semantics of a Simple Language for Parallel Programming]
- Lucid [Wadge and Ashcroft (1985): LUCID, the dataflow programming language]
- Lustre [Halbwachs, Caspi, Raymond, and Pilaud (1991): The synchronous dataflow programming language LUSTRE]
  - Clock calculus
  - 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`

ANSYS

Scade 6 and Lustre kernels



 © ANSYS, Inc.

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

The (beautiful) idea of Lustre

- Program in the specification by writing maths equations directly.
- Analyse/transform/simulate/test/verify them.
- Translate them automatically into executable code.

SCADE: Safety Critical Application Dev. Env. (Verilog, 95)

libdigital.vsw - SCADE - [RisingEdgeRetrigger/eq_RisingEdgeRetrigger]

File Edit View Node Insert Layout Project Simulink Tools Browse Window Help

libdigital.vsp
- libdigital
  - Constant Blocks
  - Variable Blocks
  - Type Blocks
  - Operators
    - count_down
    - EitherEdge
    - FallingEdge
    - FallingEdgeNoRetrigger
    - FallingEdgeRetrigger
    - FlipFlopJK
    - FlipFlopReset
    - FlipFlopSet
    - RisingEdge
    - RisingEdgeNoRetrigger
    - RisingEdgeRetrigger
      - Interface
      - eq_RisingEdgeRetrigger
    - Toggle

FileView Framework

false
FBY
RER_Output
0
count_down
PRE
RER_Input
false
0
NumberOfCycle

```
Loading project libdigital.vsp...
Constant values updated to new format
Successfully loaded project libdigital.vsp
```

Messages Dump Build Simulator

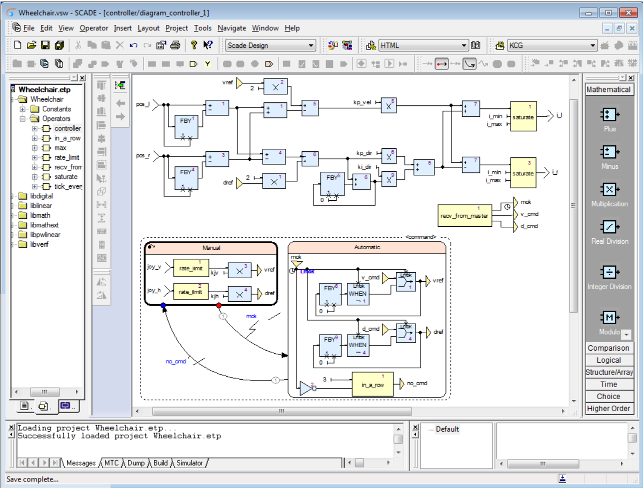
For Help, press F1

Executable Block Diagrams = "Model-Based Development"

Scade Suite — http://www.ansys.com/···

Executable Block Diagrams = “Model-Based Development”

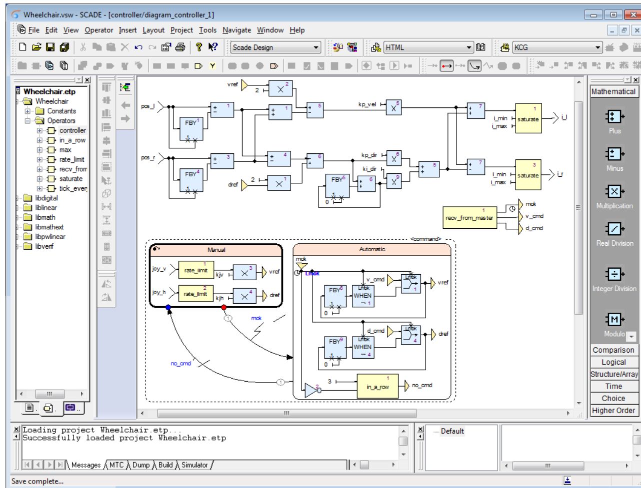
Scade Suite — http://www.ansys.com/· · ·



block	=	système
une ligne	=	signal

Executable Block Diagrams = “Model-Based Development”

Scade Suite — http://www.ansys.com/···



block = système = function on streams

une ligne = signal = stream of values

Executable Block Diagrams = "Model-Based Development"

Scade Suite — http://www.ansys.com/· · ·

```
node controller(joy_v, joy_h, pos_l, pos_r : int)
returns (i_l, i_r : int);
let
  omega_l = pos_l − (pos_l fby pos_l);
  v_err2 = (2 * v_ref) − (omega_l + omega_r);
  ...
tel
```

block	=	système	=	function on streams
une ligne	=	signal	=	stream of values

Executable Block Diagrams = “Model-Based Development”

Scade Suite — http://www.ansys.com/···

Wheelchair.vsw - SCADE - [controller/diagram_controller_1]

```
node controller(joy_v, joy_h, pos_l, pos_r : int)
returns (i_l, i_r : int);
let
  omega_l = pos_l − (pos_l fby pos_l);
  v_err2 = (2 * v_ref) − (omega_l + omega_r);
  ...
tel
```

code generator

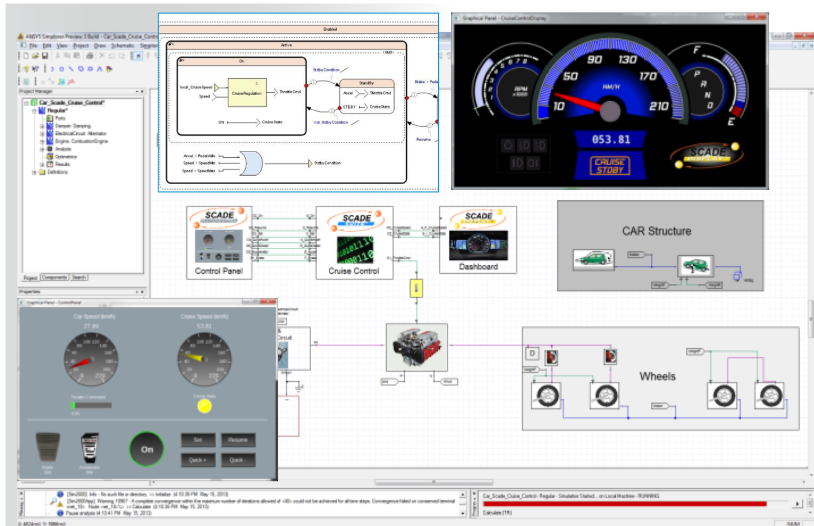
Sequential program (C, Ada, assembleur)

block	=	système	=	function on streams
une ligne	=	signal	=	stream of values

Model-Based Development: Hybrid Modelling

Embedded software interacts with physical devices

The whole system has to be modeled: the controller and the plant



(Screen shot of Simplorer + SCADE Suite: ANSYS/Esterel Technologies)

http://zelus.di.ens.fr

Compiler

Zélus is a synchronous language extended with Ordinary Differential Equations (ODEs) to model systems with complex interaction between discrete-time and continuous-time dynamics. It shares the basic principles of Lustre with features from Lucid Synchrone (type inference, hierarchical automata, and signals). The compiler is written

Research

Zélus is used to experiment with new techniques for building hybri modelers like Simulink/Stateflow and Modelica on top of a synchronou language. The language exploits novel techniques for defining th semantics of hybrid modelers, it provides dedicated type systems t ensure the absence of discontinuities during integration and th

Approach: add ODEs to a synchronous dataflow language

Reuse existing tools and techniques

Synchronous languages (SCADE/Lustre)

Expressive language for both discrete controllers and mode changes.

Off-the-shelf ODEs numeric solvers

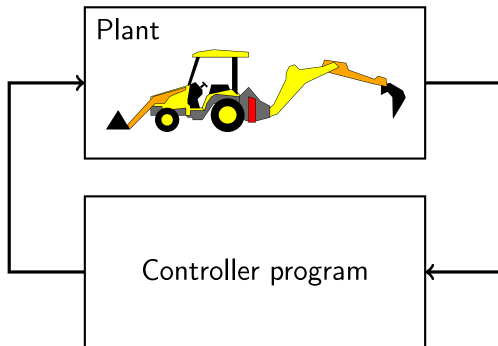
Simulate with an external, off-the-shelf, variable-step numerical solver (Sundials CVODE, [Hindmarsh, Brown, Grant, Lee, Serban, Shumaker, and Woodward (2005): SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers]).

Two concrete reasons

- Increase modeling power *(hybrid programming)*.
- Exploit existing compiler *(target for code generation)*.

Conservative: any synchronous program must be compiled, optimized, and executed as per usual.

Backhoe example



- Intended for teaching reactive programming.
- The *controller* is a regular synchronous program.
- The *plant* dynamics are modeled using hybrid automata and first-order ODEs.
- The two interact via boolean flows and signals.

Zélus: basic functions

```
let boom_rate = 0.4
```

```
val boom_rate : float
```

```
let inc x = x + 1
```

```
val inc : int -A-> int
```

Programming the controller...

Zélus: discrete programs (Controller)

```
let node after (n, t) = (c = n) where
  rec c = 0 fby min ((if t then c + 1 else c), n)
```

```
val after : int × bool -D-> bool
```

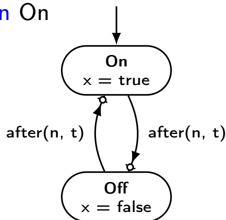
Zélus: discrete programs (Controller)

```
let node after (n, t) = (c = n) where
  rec c = 0 fby min ((if t then c + 1 else c), n)
```

```
val after : int × bool -D-> bool
```

```
let node blink (n, t) = x where
  automaton
  | On  → do x = true  until (after (n, t)) then Off
  | Off → do x = false until (after (n, t)) then On
```

```
val blink : int × bool -D-> bool
```



Cancelle

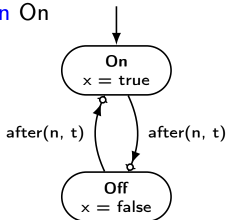
Zélus: discrete programs (Controller)

```
let node after (n, t) = (c = n) where
  rec c = 0 fby min ((if t then c + 1 else c), n)
```

```
val after : int × bool -D-> bool
```

```
let node blink (n, t) = x where
  automaton
  | On  → do x = true  until (after (n, t)) then Off
  | Off → do x = false until (after (n, t)) then On
```

```
val blink : int × bool -D-> bool
```



Cancelle

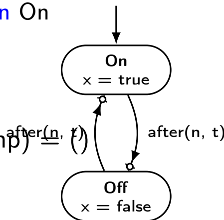
Zélus: discrete programs (Controller)

```
let node after (n, t) = (c = n) where
  rec c = 0 fby min ((if t then c + 1 else c), n)
```

```
val after : int × bool -D-> bool
```

```
let node blink (n, t) = x where
  automaton
  | On  → do x = true  until (after (n, t)) then Off
  | Off → do x = false until (after (n, t)) then On
```

```
val blink : int × bool -D-> bool
```




```
let show (alarmlamp, donelamp, cancellamp) = ()
```

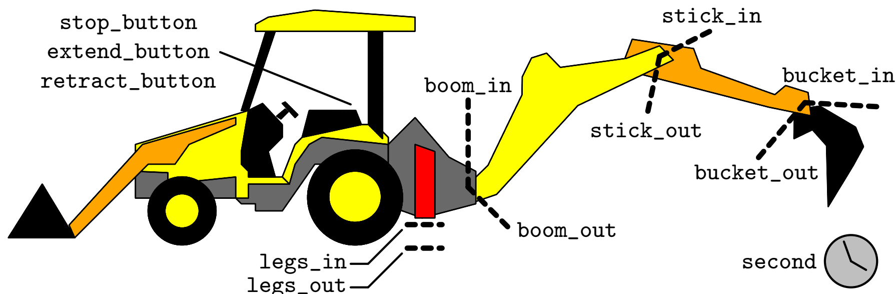
```
let node main second =
  let alarm = blink (3, second) in
  show (alarm, false, false)
```

```
val main : bool -D-> unit
```

Cancelle

Backhoe: sensors and actuators

Sensors (plant outputs / controller inputs)



Actuators (plant inputs / controller outputs)

alarm_lamp	legs_extend	boom_pull	stick_pull	bucket_pull
done_lamp	legs_retract	boom_push	stick_push	bucket_push
cancel_lamp	legs_stop	boom_drive	stick_drive	bucket_drive

(pull = in; push = out)

Modeling the plant. . .

Zélus: hybrid programs (Plant)

```
let hybrid integrator(yd, y0) = y where
  der y = yd init y0
```

```
val integrator : float × float -C-> float
```

Zélus: hybrid programs (Plant)

```
let hybrid integrator(yd, y0) = y where
  der y = yd init y0

val integrator : float × float -C-> float


let hybrid pi_controller(v_r, i) = angle where
  rec der angle = v init i
  and der v = k_p *. error +. k_i *. z init 0.0
  and der z = error init 0.0
  and error = v_r -. v

val pi_controller : float × float -C-> float
```

$$angle(t) = i + \int_0^t v(\tau)\ .d\tau$$

$$v(t) = 0 + \int_0^t (k_p(v_r(\tau) - v(\tau)) + k_i z(\tau))\ .d\tau$$

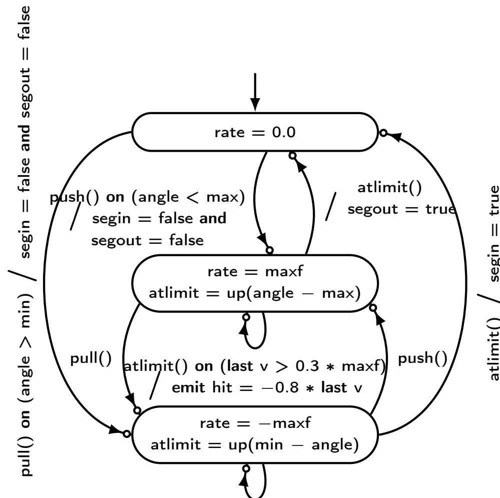
$$z(t) = 0 + \int_0^t (v_r(\tau) - v(\tau))\ .d\tau$$

Zélus: hybrid programs (Plant)

```
let hybrid integrator(yd, y0) = y where
  der y = yd init y0

val integrator : float × float -C-> float


let hybrid pi_controller(v_r, i, hit) = angle where
  rec der angle = v init i
  and der v = k_p *. error +. k_i *. z init 0.0 reset hit(v0) → v0
  and der z = error init 0.0 reset hit(_) → 0.0
  and error = v_r −. v

val pi_controller : float × float × float signal -C-> float

  present up(angle −. max) → do
    emit hit = −0.8 *. last v
  done
```

Modeling a segment

```
let hybrid segment ((min, max, i), maxf, (push, pull, go))
 = ((segin, segout), angle) where

 rec der angle = v init i
 and error = v_r −. v
 and der v = (0.7 /. maxf) ∗. error +. 0.3 ∗. z init 0.0
            reset hit(v0) → v0
 and der z = error init 0.0 reset hit(_) → 0.0
 and v_r = if go then rate else 0.0

 and init segin = angle <= min
 and init segout = angle >= max

 and automaton
   | Stuck → do rate = 0.0
     until push() on (angle < max) then
       do segin = false and segout = false in Pushing
     else  pull() on (angle > min) then
       do segin = false and segout = false in Pulling

   | Pushing → local atlimit in
     do rate = maxf and atlimit = up(angle −. max)
     until atlimit on (last v > 0.3 ∗. maxf) then do
              emit hit = −0.8 ∗. last v in Pushing
     else (atlimit) then do segout = true in Stuck
     else pull() then Pulling

   | Pulling → local atlimit in
     do rate = −. maxf and atlimit = up(min −. angle)
     until atlimit on (last v < −0.3 ∗. maxf) then do
              emit hit = −0.8 ∗. last v in Pulling
     else  (atlimit) then do segin = true in Stuck
     else  push() then Pushing
```



Finishing the Plant model

```
let hybrid model (boom_ctl, stick_ctl, bucket_ctl, leg_ctl, lamp_ctl) =

  let  (boom_inout, boom) = segment (boom_range, boom_rate, boom_ctl)
  and (stick_inout, stick) = segment (stick_range, stick_rate, stick_ctl)
  and (bucket_inout, bucket) = segment (bucket_range, bucket_rate, bucket_ctl)

  and (leg_inout, leg_pos) = legsegment (leg_range, leg_rate, leg_ctl)

  in
  (leg_inout, boom_inout, stick_inout, bucket_inout)
```

Putting the two together. . .

Composing Controller and Plant

```
let hybrid main () = () where
  rec init sensors = ((false, false), (false, false), (false, false), (false, false))

  and init (boom_drive, stick_drive, bucket_drive) = (false, false, false)
  and init (alarm_lamp, done_lamp, cancel_lamp) = (false, false, false)

  and present sample → do
      (boom_ctl, stick_ctl, bucket_ctl, legs_ctl, lamps) =
                 sampled_controller (last sensors, buttons (), second)
    done
  and sensors = model (boom_ctl, stick_ctl, bucket_ctl, legs_ctl, lamps)

  and sample = period (0.5)
  and second = present sample → not (last second) init true
```

Questions answered in the remainder of this course

- How to statically check that a program is properly initialised, i.e., that the result never depends on `nil`?
- How to statically check that combinations of fast and slow processes can execute in bounded memory?
- How to analyze a program statically and modularly to reject instantaneous feedback loops?
- How to compile a program into efficient sequential code?
- How to specify and ensure the correctness of the compilation process?
- How to model and verify distributed real-time controllers?

References I

- André, C. (Oct. 1995). SyncCharts*: A Visual Representation of Reactive Behaviors*. Technical Report. RR 95-52. Sophia-Antipolis, France: I3S.
- Berry, G. (May 1989). *Programming a Digital Watch in Esterel v3*. Rapport de recherche 1032. Sophia Antipolis: Inria.
- — (July 2000). *The Esterel v5 Language Primer*. 5.91. Ecole des Mines and INRIA.
- 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. 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. Orgun and E. Ashcroft. Sydney, Australia: World Scientific.

References II

- Champion, A., A. Mebsout, C. Sticksel, and C. Tinelli (July 2016). "The Kind 2 Model Checker". In: *Proc. 28th Int. Conf. on Computer Aided Verification (CAV 2016), Part II*. Ed. by S. Chaudhuri and A. Farzan. Vol. 9780. LNCS. Toronto, Canada: Springer, pp. 510–517.
- 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.
- — (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.
- Developers, H. (Apr. 2017). *Heptagon/BZR manual*.
- Dvorak (ed.), D. L. (Mar. 2009). *NASA Study on Flight Software Complexity*. Final Report. NASA Office of Chief Engineer.

References III

- Gérard, L., A. Guatto, C. Pasteur, and M. Pouzet (June 2012). "A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler". In: *Proc. 13th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2012)*. Ed. by R. Wilhelm, H. Falk, and W. Yi. Beijing, China: ACM Press, pp. 51–60.
- 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.
- 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., 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.

References IV

- 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.
- Halbwachs, N. and P. Raymond (Aug. 2007). *A Tutorial of Lustre*. Verimag. Gières, France.
- Hamon, G. and M. Pouzet (Sept. 2000). "Modular Resetting of Synchronous Data-Flow Programs". In: *Proc. 2nd ACM SIGPLAN Int. Conf. on Principles and Practice of Declarative Programming (PPDP 2000)*. Ed. by F. Pfenning. Montreal, Canada: ACM, pp. 289–300.
- Harel, D. (June 1987). "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3, pp. 231–274.
- Hindmarsh, A. C., P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward (Sept. 2005). "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers". In: *ACM Trans. Mathematical Software* 31.3, pp. 363–396.

References V

- 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. Int. Federation for Information Processing (IFIP) Congress 1974*. Ed. by J. L. Rosenfeld. Stockholm, Sweden: North-Holland, pp. 471–475.
- 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.

References VI

- Maraninchi, F. and Y. Rémond (2003). "Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems". In: *Science of Computer Programming* 46.3, pp. 219–254.
- Maraninchi, F. and Y. Rémond (2001). "Argos: an automaton-based synchronous language". In: *Computer Languages* 27.1–3, pp. 61–92.
- Plaice, J. A. (1988). "Sémantique et compilation de LUSTRE, un langage déclaratif synchrone". PhD thesis. Grenoble INP.
- 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.
- Raymond, P. (1991). "Compilation efficace d'un langage déclaratif synchrone: le générateur de code Lustre-V3". PhD thesis. Grenoble INP.

References VII

- 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.