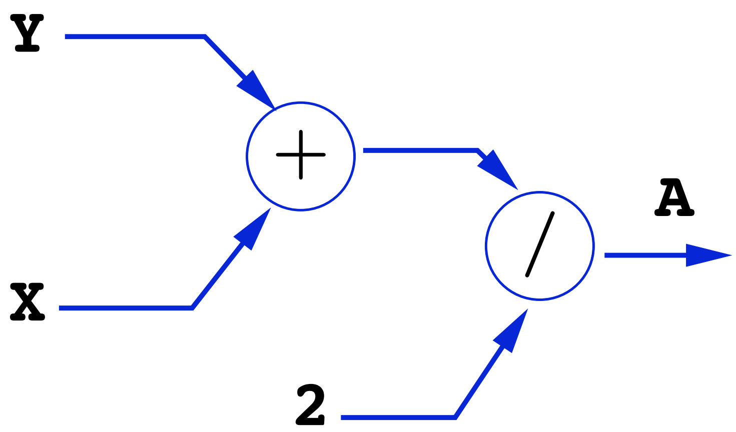
The Lustre Language

Synchronous Programming

Pascal Raymond, Nicolas Halbwachs
Verimag-CNRS

Data-flow approach

- A program = a network of operators connected by wires
- Rather classical (control theory, circuits)




```
node Average(X, Y : int)
returns (A : int);
let
    A = (X + Y) / 2 ;
tel
```

- Synchronous: discrete time $= \mathbb{N}$

  $\forall t \in \mathbb{N} \;\; A_t = (X_t + Y_t)/2$

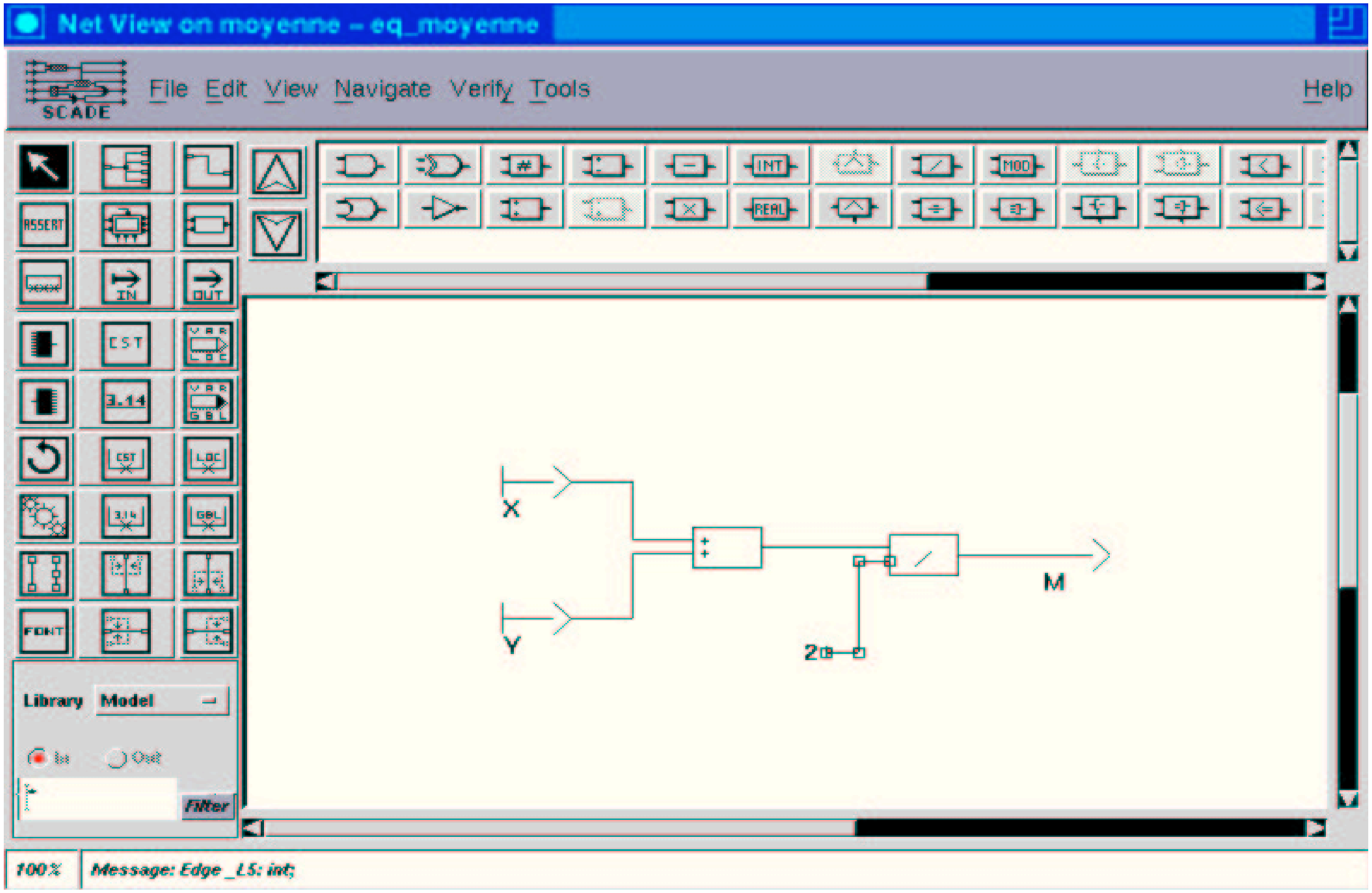
- Full parallelism: nodes are running concurrently

Another version

```
node Average(X, Y : int)
returns (A : int);
var S : int;          -- local variable
let
   A = S / 2;         -- equations
   S = X + Y;         -- (order does not matter)
tel
```

- **declarative: set of equations**
- **a single equation for each output/local**
- **variables are infinite sequences of values**

Lustre (textual) and Scade (graphical)



Combinational programs

- **Basic types: bool, int, real**

- **Constants:**

  $\mathbf{2} \equiv 2, 2, 2, \cdots$

  $\mathtt{true} \equiv true, true, true, \cdots$

- **Pointwise operators:**

  $\mathtt{X} \equiv \; x_0, x_1, x_2, x_3 ... \quad \mathtt{Y} \equiv \; y_0, y_1, y_2, y_3 ...$

  $\mathtt{X \; + \; Y} \equiv x_0 + y_0, \; x_1 + y_1, \; x_2 + y_2, \; x_3 + y_3 ...$

- **All classical operators are provided**

- `if` operator

```
node Max(A,B: real) returns (M: real);
let
   M = if (A >= B) then A else B;
tel
```

Warning: functional "`if then else`", not statement

```
let
  if (A >= B) then M = A ;
  else M = B ;
tel
```

Memory programs

Delay operator

- **Previous operator: `pre`**

X	x_0	x_1	x_2	x_3	x_4	...
`pre` X	nil	x_0	x_1	x_2	x_3	...

Memory programs

Delay operator

- **Previous operator: `pre`**

X	x_0	x_1	x_2	x_3	x_4	...
`pre` X	nil	x_0	x_1	x_2	x_3	...

i.e. $(\texttt{pre} X)_0$ undefined and $\forall i \neq 0 \ (\texttt{pre} X)_i = X_{i-1}$

Memory programs

Delay operator

- **Previous operator: `pre`**

X	x_0	x_1	x_2	x_3	x_4	...
`pre` X	nil	x_0	x_1	x_2	x_3	...

i.e. $(\texttt{pre}X)_0$ undefined and $\forall i \neq 0 \ (\texttt{pre}X)_i = X_{i-1}$

- **Initialization: `->`**

X	x_0	x_1	x_2	x_3	x_4	...
Y	y_0	y_1	y_2	y_3	y_4	...
X ->Y	x_0	y_1	y_2	y_3	y_4	...

Memory programs

Delay operator

- **Previous operator: `pre`**

X	x_0	x_1	x_2	x_3	x_4	...
`pre` X	nil	x_0	x_1	x_2	x_3	...

i.e. $(\texttt{pre}X)_0$ undefined and $\forall i \neq 0\ (\texttt{pre}X)_i = X_{i-1}$

- **Initialization: `->`**

X	x_0	x_1	x_2	x_3	x_4	...
Y	y_0	y_1	y_2	y_3	y_4	...
X ->Y	x_0	y_1	y_2	y_3	y_4	...

i.e. $(X\texttt{->}Y)_0 = X_0$ and $\forall i \neq 0\ (X\texttt{->}Y)_i = Y_i$

Nodes with memory

- **Boolean example: raising edge**

```
node Edge (X : bool) returns (E : bool);
let
   E = false -> X and not pre X ;
tel
```

- **Numerical example: min and max of a sequence**

```
node MinMax(X : int)
returns (min, max :  int); – several outputs
let
   min = X -> if (X < pre min) then X else pre min;
   max = X -> if (X > pre max) then X else pre max;
tel
```

Recursive definition

Examples

- `N = 0 -> pre N + 1`

Recursive definition

Examples

- `N = 0 -> pre N + 1` $N = 0, 1, 2, 3, \cdots$

Recursive definition

Examples

- `N = 0 -> pre N + 1` $N = 0, 1, 2, 3, \cdots$
- `A = false -> not pre A`

Recursive definition

Examples

- `N = 0 -> pre N + 1` $N = 0, 1, 2, 3, \cdots$
- `A = false -> not pre A` $A =$ *false*, *true*, *false*, *true*, $\cdots$

Recursive definition

Examples

- `N = 0 -> pre N + 1` $N = 0, 1, 2, 3, \cdots$
- `A = false -> not pre A` $A =$ *false*, *true*, *false*, *true*, $\cdots$
- Correct $\Rightarrow$ the sequence can be computed step by step

Counter-example

- `X = 1/(2-X)`
- unique (integer) solution: "X=1"
- but not computable step by step

Sufficient condition: forbid combinational loops

How to detect combinational loops?

Syntactic vs semantic loop

- Example:

```
X = if C then Y else A;
Y = if C then B else X;
```

- Syntactic loop
- But not semantic: X = Y = if C then B else A

Correct definitions in Lustre

- Choice: syntactic loops are rejected

  (even if they are “false” loops)

Exercices

- A flow $F = 1, 1, 2, 3, 5, 8, \cdots$?
- A node `Switch(on,off: bool) returns (s: bool);` such that:
  - ⋆ `s` raises (*false* to *true*) if `on`, and falls (*true* to *false*) if `off`
  - ⋆ everything behaves as if `s` was *false* at the origin
  - ⋆ must work properly even if `off` and `on` are the same
- A node `Count(reset,x: bool) returns (c: int);` such that:
  - ⋆ `c` is reset to 0 if `reset`, otherwise it is incremented if `x`,
  - ⋆ everything behaves as if `c` was 0 at the origin

Solutions

- **Fibonacci:**

```
f = 1 -> pre( f + (0 -> pre f));
```

- **Bistable:**

```
node Switch(on,off: bool) returns (s: bool);
let s = if(false -> pre s) then not off else on; tel
```

- **Counter:**

```
node Count(reset,x: bool) returns (c: int);
let
   c = if reset then 0
      else if x then (0->pre c) + 1
      else (0->pre c);
tel
```

Modularity

Reuse

- Once defined, a user node can be used as a basic operator
- Instanciation is functional-like
- Example (exercice: what is the value?)

```
A = Count(true -> (pre A = 3), true)
```

- Several outputs:

```
node MinMaxAverage(x: bool) returns (a: int);
var min,max: int;
let
   a = average(min,max);
   min, max = MinMax(x);
tel
```

A complete example: stopwatch

- 1 integer output: displayed `time`
- 3 input buttons: `on_off, reset, freeze`
  - ⋆ `on_off` starts and stops the stopwatch
  - ⋆ `reset` resets the stopwatch (if not running)
  - ⋆ `freeze` freezes the displayed time (if running)

A complete example: stopwatch

- 1 integer output: displayed `time`
- 3 input buttons: `on_off, reset, freeze`
  - ⋆ `on_off` starts and stops the stopwatch
  - ⋆ `reset` resets the stopwatch (if not running)
  - ⋆ `freeze` freezes the displayed time (if running)
- Find local variables (and how they are computed):
  - ⋆ `running:  bool`, a *Switch* instance
  - ⋆ `freezed:  bool`, a *Switch* instance
  - ⋆ `cpt:  int`, a *Count* instance

```
node Stopwatch(on_off,reset,freeze: bool)
returns (time: int);
var running, freezed:bool; cpt:int;
let
   running = Switch(on_off, on_off);
   freezed = Switch(
             freeze and running,
             freeze or on_off);
   cpt = Count(reset and not running, running);
   time = if freezed then (0 -> pre time) else cpt;
tel
```

Clocks

Motivation

- Attempt to conciliate “control” with data-flow
- Express that some part of the program works *less often*
- $\Rightarrow$ notion of data-flow clock (similar to clock-enabled in circuit)

Clocks

Motivation

- Attempt to conciliate "control" with data-flow
- Express that some part of the program works *less often*
- $\Rightarrow$ notion of data-flow clock (similar to clock-enabled in circuit)

Sampling: `when` operator

`X`	4	1	-3	0	2	7	8
`C`	*true*	*false*	*false*	*true*	*true*	*false*	*true*
`X when C`	4			0	2		8

- whenever `C` is false, `X when C` does not exist

Projection: `current` operator

- One can operate only on flows with the same clock
- projection on a common clock is (sometime) necessary

`X`	4	1	-3	0	2	7	8
`C`	*true*	*false*	*false*	*true*	*true*	*false*	*true*
`Y = X when C`	4			0	2		8
`Z = current(Y)`	4	4	4	0	2	2	8

Nodes and clocks

- Clock of a node instance = clock of its effective inputs
- Sampling inputs = enforce the whole node to run slower
- In particular, sampling inputs $\neq$ sampling outputs

`C`	*true*	*true*	*false*	*false*	*true*	*false*	*true*
`Count((r,true) when C)`	1	2			3		4
`Count(r,true) when C`	1	2			5		7

Example: stopwatch with clocks

```
node Stopwatch(on_off,reset,freeze: bool)
returns (time: int);
var running, freezed:bool;
   cpt_ena, tim_ena :  bool;
   (cpt:int) when cpt_ena;
let
   running = Switch(on_off, on_off);
   freezed = Switch(
             freeze and running,
             freeze or on_off);
   cpt_ena = true -> reset or running;
   cpt = Count((not running, true) when cpt_ena);
   tim_ena = true -> not freezed;
   time = current(current(cpt) when tim_ena);
tel
```

Clock checking

- Similar to type checking
- Clocks must be named (clocks are equal iff they are the same var)
- The clock of each var must be declared (the default is the base clock)
- $clk(\texttt{exp when C}) = \texttt{C} \Leftrightarrow clk(\texttt{exp}) = clk(\texttt{C})$
- $clk(\texttt{current exp}) = clk(clk(\texttt{exp}))$
- For any other `op`:
  $clk(\texttt{e1 op e2}) = \texttt{C} \Leftrightarrow clk(\texttt{e1}) = clk(\texttt{e2}) = \texttt{C}$

Programming with clocks

- Clocks are the right semantic solution
- However, using clocks is quite tricky (cf. stopwatch)
- Main problem: initialisation

  `current(X when C)` exists, but is undefined until `C` becomes true for the first time
- Solution: *activation condition*
  - ⋆ not an operator, rather a *macro*
  - ⋆ `X = CONDACT(OP, clk, args, dflt)` equivalent to:

    ```
    X = if clk then current(OP(args when clk))
        else (dflt -> pre X)
    ```
  - ⋆ Provided by Scade (industrial)

Is that all there is?

Dedicated vs general purpose languages

- Synchronous languages are dedicated to *reactive kernel*
- Not suitable for complex data types manupulation
- Abstract types and functions are *imported* from the host language (typically C)

However ...

- Statically sized arrays are provided
- Static recursion (Lustre V4, dedicated to circuit)
- Modules and templates (Lustre V6, dedicated to sofware)