Lustre: adding Arrays, Resets, and State Machines

Timothy.Bourke@inria.fr

and

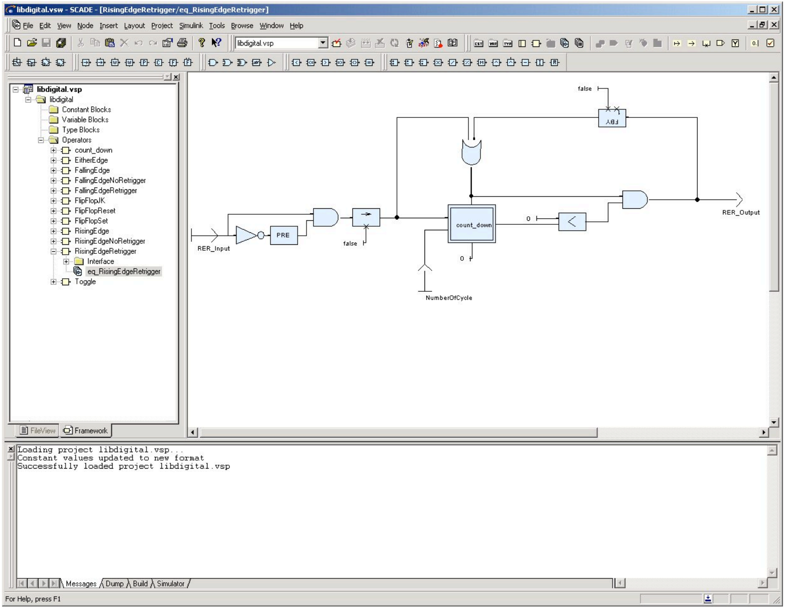
Marc.Pouzet@ens.psl.eu

MPRI 2.23.1: Synchronous Reactive Systems, 17 September 2024

The (beautiful) idea of Lustre

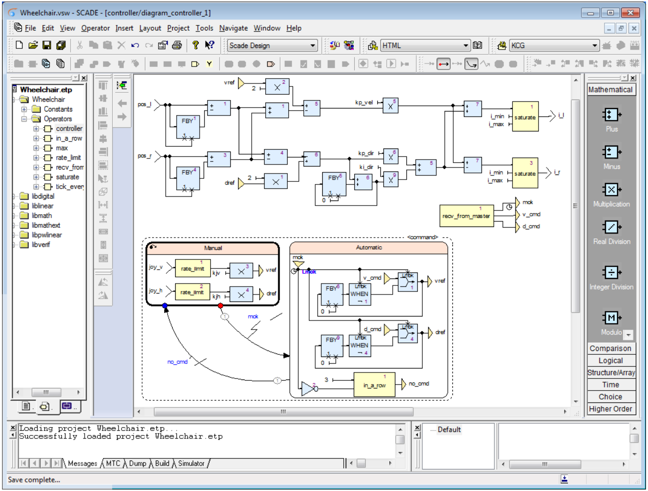
- Program in the specification by writing maths equations directly.
- Analyse/transform/simulate/test/verify them.
- Translate them automatically into executable code.

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



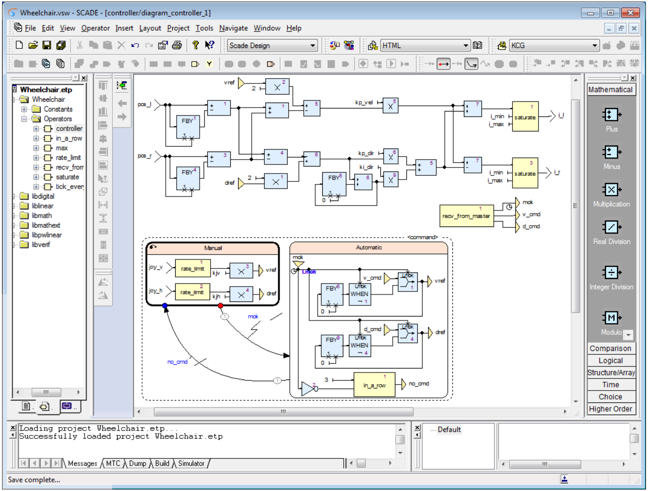
Executable Block Diagrams = "Model-Based Development"

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



Executable Block Diagrams = "Model-Based Development"

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

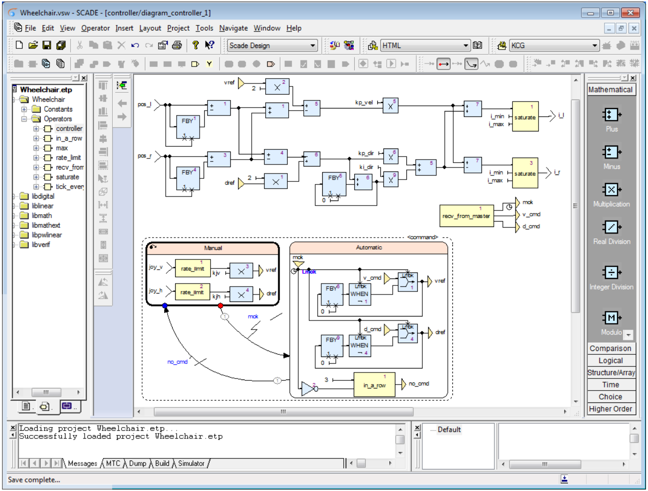


block = system

line = signal

Executable Block Diagrams = "Model-Based Development"

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

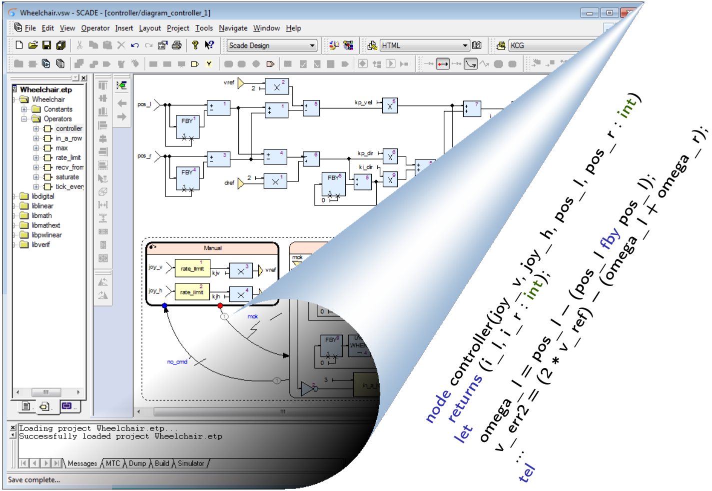


block = system = function on streams

line = signal = stream of values

Executable Block Diagrams = "Model-Based Development"

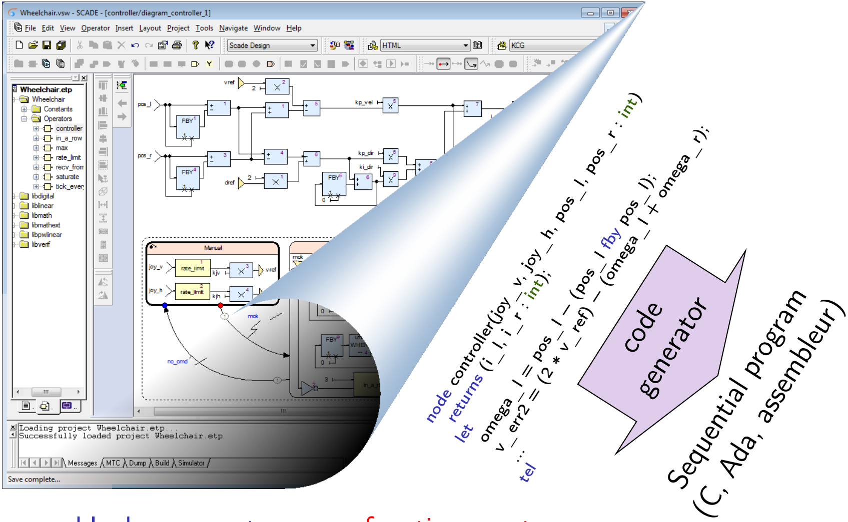
Scade Suite — http://www.ansys.com/· · ·



block	=	system	=	function on streams
line	=	signal	=	stream of values

Executable Block Diagrams = "Model-Based Development"

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



block = system = function on streams

line = signal = stream of values

Introduction

Arrays in Lustre

Modular reset

State machines

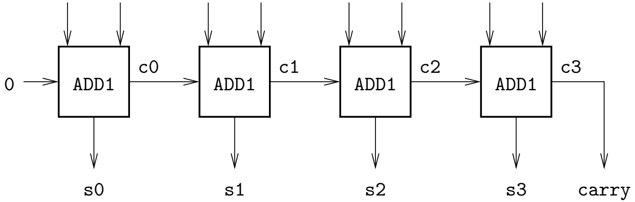
Conclusion

Arrays

- As usual, an array is an indexed collection of homogeneous elements.
- In Lustre: mix streams and arrays.
  - Repeated structures, e.g., *n*-bit adder.
  - Efficient implementations, e.g., FIFO.
  - Matrices are important in many embedded applications.
- There are two main approaches.
  - Lustre v4: syntactic (macro) expansion to basic equations.
    [Rocheteau (1992): Extension du langage LUSTRE et application à la conception de circuits: le langage LUSTRE-V4 et le système POLLUX] [Caspi, Halbwachs, Maraninchi, Morel, and Raymond (2014): Arrays in Lustre]
  - Lustre v6/SCADE 6/Heptagon: first-class functional arrays with a fixed set of higher-order iterators
    [Morel (2007): Array Iterators in Lustre: From a Language Extension to Its Exploitation in Validation] [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]

Arrays in Lustre v4: 1/4

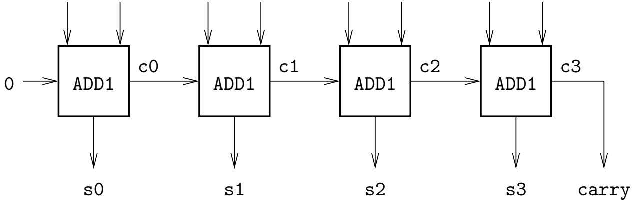
Build a 4-bit binary adder, using the full adder node from last week.



[Halbwachs and Raymond (2007): A Tutorial of Lustre]

Arrays in Lustre v4: 1/4

Build a 4-bit binary adder, using the full adder node from last week.



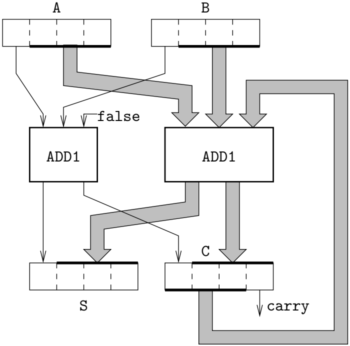
[Halbwachs and Raymond (2007): A Tutorial of Lustre]

```
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
```

- Version without arrays
- Manual instantiation and wiring
- Cannot parameterize for *n* bits

Arrays in Lustre v4: 2/4

```
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
```



- Array size is known at compile time
- Access to array elements: A[i], where i is known at compile time and $0 \leq i \leq size - 1$
- Array slices: A[i..j], where i and j are known at compile time, A[i..j] = [A[i], A[i+1], ..., A[j]] if $i \leq j$ A[i..j] = [A[i], A[i−1], ..., A[j]] if $j < i$

Arrays in Lustre v4: 3/4

```
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
```

Generalize and simplify:

```
node add (const n : int; A, B: bool^n) returns (S: bool^n; carry: bool);
var C: bool^n;
let
  (S, C) = full_add(A, B, [false] | C[0..n−2]);
  carry = C[n−1];
tel
```

Three new features:

- Node parameters (constant inputs): value is known at compile time.
- Constant arrays: e.g., [0, 1, 2]
- Concatentation: A | B is [A[0], A[1], ..., A[n−1], B[0], B[1], ..., B[m−1]]

Arrays in Lustre v4: 4/4

Basic operators are polymorphic and thus apply to arrays:

E.g., A = true^4 -> if c then B[4..7] else pre(A)

Arrays in Lustre v4: 4/4

Basic operators are polymorphic and thus apply to arrays:

E.g., A = true^4 -> if c then B[4..7] else pre(A)

Other operators are extended to operate pointwise:

E.g., A or B means [A[0] or B[0], A[1] or B[1], ..., A[n−1] or B[n−1]]

Arrays in Lustre v4: 4/4

Basic operators are polymorphic and thus apply to arrays:

E.g., A = true^4 -> if c then B[4..7] else pre(A)

Other operators are extended to operate pointwise:

E.g., A or B means [A[0] or B[0], A[1] or B[1], ..., A[n−1] or B[n−1]]

Recursive expansion of arrays to single variables at compilation.

- Useful for hardware descriptions
- Useful for verification: scalable examples, bit-blasting of integers
- Not really suitable for software.

Modern Lustre Arrays

SCADE 6, Lustre v6, and Heptagon use functional arrays:

- They are not mutable.
- Changing an element produces a new array.
- A fixed set of higher-order operators provide useful patterns.
  - Heptagon provides a subset of SCADE 6 operators: [Heptagon Developers (2017): Heptagon/BZR manual] map, mapi, fold, foldi, and mapfold.
  - Direct translation into software.
  - Avoids the problems of arbitrary dependencies within a single array, e.g.,

    x[1..2] = g(y[4..5]);
    x[3..5] = h(y[0..3]);
  - Care needed to avoid unnecessary intermediate arrays and copying.

Heptagon arrays: basic operations

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

Can combine array operations with standard operators, e.g.,

```
node test(x : int^4; i : int) returns (y : int; o : bool^4);
let
  y = x.[i] default −1;
  o = ([true, false, true, false]) fby ([not o[3]] @ o[0..2]);
tel
```

Exercise: *n*-place FIFOs

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

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

  Note the syntax for (static) node parameters.

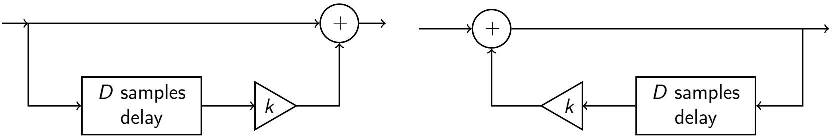
- Use it to calculate a sliding average (as for slide 35 of course 1).

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

  Instantiating a parameterized node: `fifo<<7, 0>>(x)`

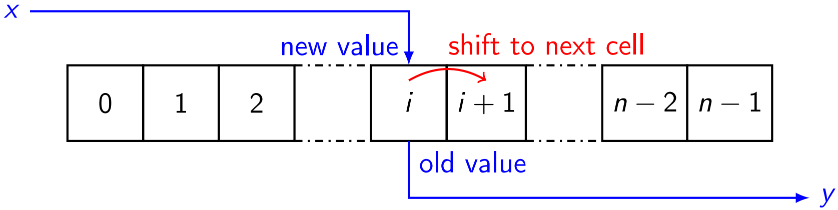
- Use it to generate echo effects on audio signals. (e.g., see `https://sound.eti.pg.gda.pl/student/eim/synteza/adamx/eindex.html`)

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

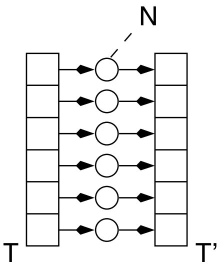


Hint: n-place FIFOs

Implement with a circular buffer.



Heptagon arrays: map

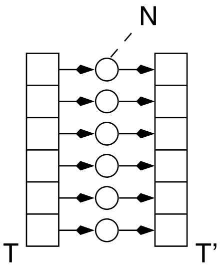



```
node f(x, y : int) returns (r : int);
let
  r = x + 10 * y + (0 fby r);
tel

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

- Must give array size when using iterator.
- Number of input and output arrays depends on the node being iterated; given f : t1 ∗ t2 ∗ t3 −> t4 ∗ t5, then map<<n>> f : t1^n ∗ t2^n ∗ t3^n −> t4^n ∗ t5^n.
- The iterated node can be stateful, e.g., contain fbys.
- map<<n>> f <(a)>(t) means ∀i, o[i] = f(a, t[i]).

Heptagon arrays: mapi

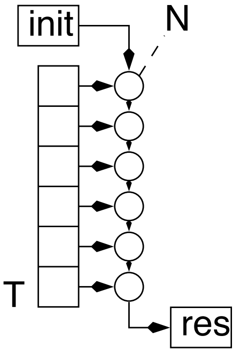



```
fun f_i(x, y, i : int) returns (r1, r2 : int);
let
  r1 = x + 10 * y;
  r2 = i;
tel

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

- mapi is a map that also passes the array index to each iterated instance.
- (The fun declares f_i as a combinatorial function.)

Heptagon arrays: fold

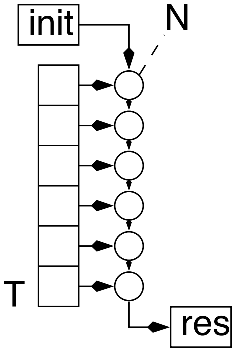



```
fun f(x, y : int) returns (r : int);
let
  r = x + 10 * y;
tel

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

- Each node acts on an array element and an accumulated value received from the previous node and updated for the next one.
- The caller initializes the accumulator and receives its final value.
- Given f : t1 * t −> t, then fold<<n>> f : t1^n * t −> t.
- fold<<2>> f(t, 0) calculates f(t[1], f(t[0], 0)).

Heptagon arrays: foldi

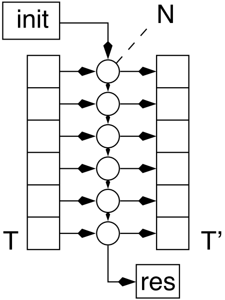



```
fun g(x, y, i, acc : int) returns (acc' : int);
let
  acc' = if (i % 2 = 0) then (x + y + acc) else acc;
tel

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

- foldi is similar but also passes each node its index.
- Iterators can be applied to multidimensional arrays; (mapi and foldi receive an index value per dimension).

Heptagon arrays: mapfold




```
fun g(x, acc : int) returns (y, acc' : int);
let
  y = 2 * (0 fby x);
  acc' = acc + y;
tel

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

- mapfold combines map and fold by accumulating a value and producing one or more new arrays.
- Given f : t1 ∗ t −> t2 ∗ t, then mapfold<<n>> f : t1^n ∗ t −> t2^n ∗ t.

Exercise: programming with array iterators

- Program an 8-bit adder.

  `node adder(a, b : bool^8) returns (s : bool^8; co : bool);`

- Track 8 incoming boolean signals and return true whenever a rising edge is detected on any of them.

  `node edgedetect(s : bool^8) returns (edge : bool);`

- Track 8 incoming boolean signals and return the count of detected edges over the last 3 cycles.

  `node edgecount(s : bool^8) returns (count : int);`

Iterators: semi-linear typing

Semi-linear typing is used 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]

Introduction

Arrays in Lustre

Modular reset

State machines

Conclusion

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
```

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F
i	0
nat(i)	0
(restart nat every r)(i)	0

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F
i	0	5
nat(i)	0	1
(restart nat every r)(i)	0	1

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T
i	0	5	10
nat(i)	0	1	2
(restart nat every r)(i)	0	1	10

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F
i	0	5	10	15
nat(i)	0	1	2	3
(restart nat every r)(i)	0	1	10	11

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F
i	0	5	10	15	20
nat(i)	0	1	2	3	4
(restart nat every r)(i)	0	1	10	11	12

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T
i	0	5	10	15	20	25
nat(i)	0	1	2	3	4	5
(restart nat every r)(i)	0	1	10	11	12	25

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F
i	0	5	10	15	20	25	30
nat(i)	0	1	2	3	4	5	6
(restart nat every r)(i)	0	1	10	11	12	25	26

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T
i	0	5	10	15	20	25	30	35
nat(i)	0	1	2	3	4	5	6	7
(restart nat every r)(i)	0	1	10	11	12	25	26	35

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F
i	0	5	10	15	20	25	30	35	40
nat(i)	0	1	2	3	4	5	6	7	8
(restart nat every r)(i)	0	1	10	11	12	25	26	35	36

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F	···
i	0	5	10	15	20	25	30	35	40	···
nat(i)	0	1	2	3	4	5	6	7	8	···
(restart nat every r)(i)	0	1	10	11	12	25	26	35	36	···

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F	···
i	0	5	10	15	20	25	30	35	40	···
nat(i)	0	1	2	3	4	5	6	7	8	···
(restart nat every r)(i)	0	1	10	11	12	25	26	35	36	···

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F	⋯
i	0	5	10	15	20	25	30	35	40	⋯
nat(i)	0	1	2	3	4	5	6	7	8	⋯
(restart nat every r)(i)	0	1	10	11	12	25	26	35	36	⋯

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F	···
i	0	5	10	15	20	25	30	35	40	···
nat(i)	0	1	2	3	4	5	6	7	8	···
(restart nat every r)(i)	0	1	10	11	12	25	26	35	36	···

A simple example with reset

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F	···
i	0	5	10	15	20	25	30	35	40	···
nat(i)	0	1	2	3	4	5	6	7	8	···
(restart nat every r)(i)	0	1	10	11	12	25	26	35	36	···

Higher-order definition of modular reset

Recursive definition (not valid in Lustre) [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs]

> f(x) every r acts as f(x) until r is true and after r has been true, [it] acts as f(x') every r' where x' (resp. r') is the sub-stream of x (resp. r) starting when r is true.

Higher-order definition of modular reset

Recursive definition (not valid in Lustre) [Hamon and Pouzet (2000): Modular Resetting of Synchronous Data-Flow Programs]

> f(x) every r acts as f(x) until r is true and after r has been true, [it] acts as f(x') every r' where x' (resp. r') is the sub-stream of x (resp. r) starting when r is true.

```
node true_until(r: bool) returns (c: bool)
let
  c = true -> if r then false else pre c;
tel

node reset_f(x: int, r: bool) returns (y: int)
  var c: bool;
let
  c = true_until(r);
  y = merge c (f(x when c)) (reset_f((x, r) whenot c));
tel
```

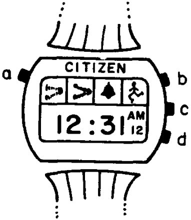


Fig. 7.

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

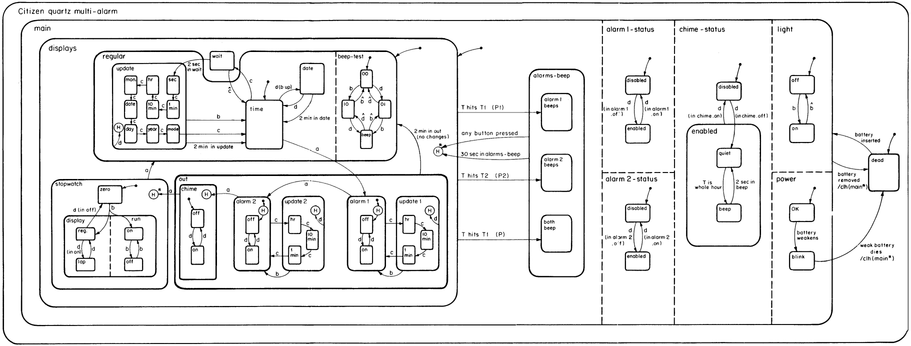


Fig. 31

Describe control-dominated systems, as opposed to data-dominated systems. Express mode-specific behaviour.

Statecharts: synchronous language approach

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]

  [Maraninchi and Halbwachs (1996): Compiling Argos into Boolean equations]

- Esterel

  [Berry (2000): The Esterel v5 Language Primer] [Berry (1989): Programming a Digital Watch in Esterel v3]

- SyncCharts → Safe State Machines

  [André (1995): SyncCharts: A Visual Representation of Reactive Behaviors]

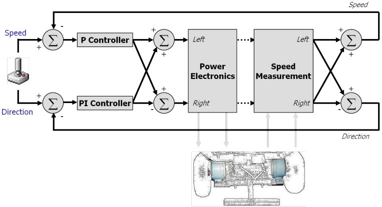
- State machines in Lucid Synchrone

  [Pouzet (2006): Lucid Synchrone, v. 3. Tutorial and reference manual] [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]

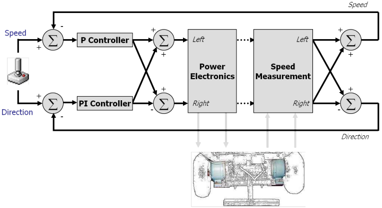
- Scade 6 State Machines

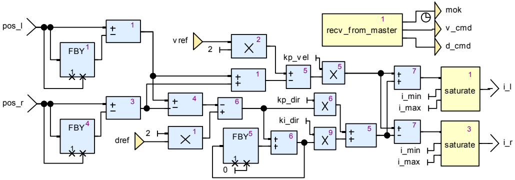
  [Colaço, Pagano, and Pouzet (2005): A Conservative Extension of Synchronous Data-flow with State Machines]

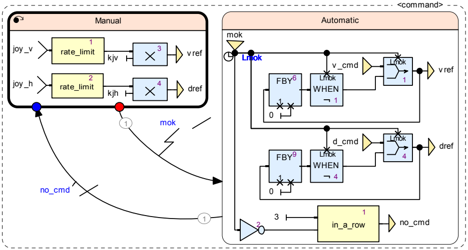
Return of the robotic wheelchair. . .



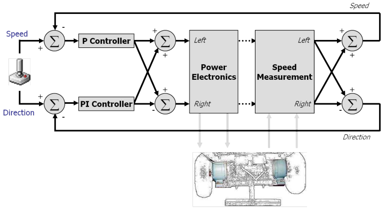
Return of the robotic wheelchair. . .

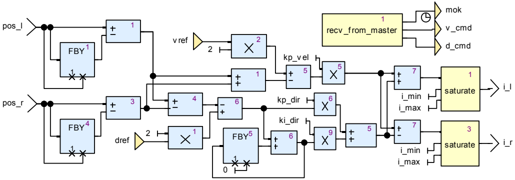


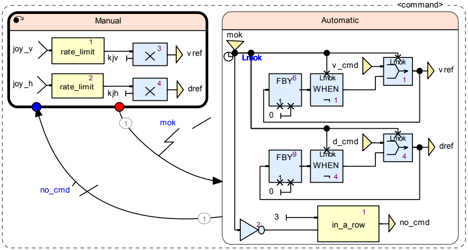




Return of the robotic wheelchair. . .








```
node controller(joyv, joyh, posl, posr : int)
returns (il, ir : int);
var vref, dref, ...;
let
  omegal = posl − (posl fby posl);
  omegar = posr − (posr fby posr);
  verr = (2 ∗ vref) − (omegal + omegar);
  derr = (2 ∗ dref) − (omegal − omegar);
  ivel = kpvel ∗ verr;
  idir = kpdir ∗ derr + kidir ∗ derr2;
  derr2 = (0 fby derr2) + derr;
  il = saturate(ivel + idir, imin, imax);
  ir = saturate(ivel − idir, imin, imax);

  (mok, vcmd, dcmd) = recvfrommaster();
  automaton
    state Manual do
      vref = kjv ∗ ratelimit(joyv);
      dref = kjh ∗ ratelimit(joyh);
    unless mok then Automatic

    state Automatic do
      vref = merge mok vcmd
                    ((0 fby vref) when not mok);
      dref = merge mok dcmd
                    ((0 fby dref) when not mok);
    until in_a_row(3, not mok) then Manual
  end;
tel
```

Two modes: manual (joystick) and automatic (command from master)

State Machines

A node is defined by a list of definitions: each is an equation or a state machine.

A state machine is a list of named states:

- Exactly one is active in any cycle; (starting with the first state).
- Each contains itself a list of definitions, that may include other state machines.
- A variable defined by a state machine must be given a value in each state (possibly implicitly).

A state includes a list of transitions.

- Transition = guard + destination state.
- If guard is true, active state changes.
- Transitions are evaluated in order.

```
node controller(joyv, joyh, posl, posr : int)
returns (il, ir : int);
var vref, dref, ...;
let
  omegal = posl − (posl fby posl);
  omegar = posr − (posr fby posr);
  verr = (2 ∗ vref) − (omegal + omegar);
  derr = (2 ∗ dref) − (omegal − omegar);
  ivel = kpvel ∗ verr;
  idir = kpdir ∗ derr + kidir ∗ derr2;
  derr2 = (0 fby derr2) + derr;
  il = saturate(ivel + idir, imin, imax);
  ir = saturate(ivel − idir, imin, imax);

  (mok, vcmd, dcmd) = recvfrommaster();
  automaton
    state Manual do
      vref = kjv ∗ ratelimit(joyv);
      dref = kjh ∗ ratelimit(joyh);
    unless mok then Automatic

    state Automatic do
      vref = merge mok vcmd
                 ((0 fby vref) when not mok);
      dref = merge mok dcmd
                 ((0 fby dref) when not mok);
    until in_a_row(3, not mok) then Manual
  end;
tel
```

Transitions: weak and strong

For any state machine, only one set of equations is active in each reaction.

Is a transition taken *before* deciding which equations are active, or *after* having evaluated the active equations?

Transitions: weak and strong

For any state machine, only one set of equations is active in each reaction.

Is a transition taken *before* deciding which equations are active, or *after* having evaluated the active equations?

Both cases are useful and possible.

- strong preemption: ... unless g then S

  The guard is tested and the transition taken before determining the active state (and equations).

  The guard must be stateless (e.g., no fbys) and cannot refer to variables defined within the state.

Transitions: weak and strong

For any state machine, only one set of equations is active in each reaction.

Is a transition taken *before* deciding which equations are active, or *after* having evaluated the active equations?

Both cases are useful and possible.

- strong preemption: ... unless g then S

  The guard is tested and the transition taken before determining the active state (and equations).

  The guard must be stateless (e.g., no fbys) and cannot refer to variables defined within the state.

- weak preemption: ... until g then S

  The guard is tested after evaluating the equations in the active state.

  The new state is active in the next cycle unless strongly preempted.

  The guard may be stateful and refer to variables within the state.

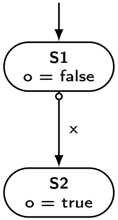
Strong transitions

Simple state machine: o = false up until the instant that x becomes true, then o = true immediately and henceforth.

Strong preemption: o becomes true at the instant that x does.

```
automaton
  state S1 do
    o = false
  unless x then S2

  state S2 do
    o = true
end
```



(The SCADE 6 ‘bubble before’ notation suggests that the transition and destination state activate together in a reaction.)

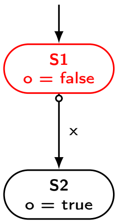
Strong transitions

Simple state machine: o = false up until the instant that x becomes true, then o = true immediately and henceforth.

Strong preemption: o becomes true at the instant that x does.

```
automaton
  state S1 do
    o = false
  unless x then S2

  state S2 do
    o = true
end
```



x	F
o	F

(The SCADE 6 ‘bubble before’ notation suggests that the transition and destination state activate together in a reaction.)

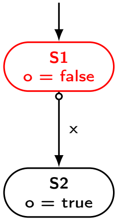
Strong transitions

Simple state machine: o = false up until the instant that x becomes true, then o = true immediately and henceforth.

Strong preemption: o becomes true at the instant that x does.

```
automaton
  state S1 do
    o = false
  unless x then S2

  state S2 do
    o = true
end
```



x	F	F
o	F	F

(The SCADE 6 ‘bubble before’ notation suggests that the transition and destination state activate together in a reaction.)

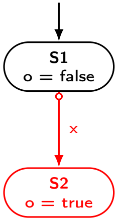
Strong transitions

Simple state machine: o = false up until the instant that x becomes true, then o = true immediately and henceforth.

Strong preemption: o becomes true at the instant that x does.

```
automaton
  state S1 do
    o = false
  unless x then S2

  state S2 do
    o = true
end
```



x	F	F	T
o	F	F	T

(The SCADE 6 ‘bubble before’ notation suggests that the transition and destination state activate together in a reaction.)

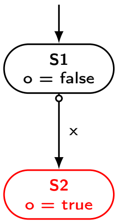
Strong transitions

Simple state machine: o = false up until the instant that x becomes true, then o = true immediately and henceforth.

Strong preemption: o becomes true at the instant that x does.

```
automaton
  state S1 do
    o = false
  unless x then S2

  state S2 do
    o = true
end
```



x	F	F	T	F
o	F	F	T	T

(The SCADE 6 ‘bubble before’ notation suggests that the transition and destination state activate together in a reaction.)

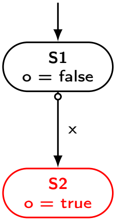
Strong transitions

Simple state machine: o = false up until the instant that x becomes true, then o = true immediately and henceforth.

Strong preemption: o becomes true at the instant that x does.

```
automaton
  state S1 do
    o = false
  unless x then S2

  state S2 do
    o = true
end
```



x	F	F	T	F	F
o	F	F	T	T	T

(The SCADE 6 ‘bubble before’ notation suggests that the transition and destination state activate together in a reaction.)

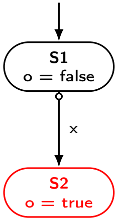
Strong transitions

Simple state machine: o = false up until the instant that x becomes true, then o = true immediately and henceforth.

Strong preemption: o becomes true at the instant that x does.

```
automaton
  state S1 do
    o = false
  unless x then S2

  state S2 do
    o = true
end
```



x	F	F	T	F	F	T
o	F	F	T	T	T	T

(The SCADE 6 ‘bubble before’ notation suggests that the transition and destination state activate together in a reaction.)

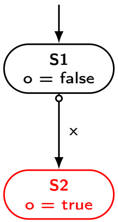
Strong transitions

Simple state machine: o = false up until the instant that x becomes true, then o = true immediately and henceforth.

Strong preemption: o becomes true at the instant that x does.

```
automaton
  state S1 do
    o = false
  unless x then S2

  state S2 do
    o = true
end
```



x	F	F	T	F	F	T	⋯
o	F	F	T	T	T	T	⋯

(The SCADE 6 ‘bubble before’ notation suggests that the transition and destination state activate together in a reaction.)

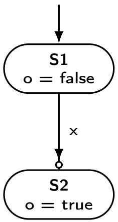
Weak transitions

Simple state machine: o = false until x becomes true, then o = true from the next instant onwards.

Weak preemption: o becomes true the instant after x does.

```
automaton
  state S1 do
    o = false
  until x then S2

  state S2 do
    o = true
end
```



(The SCADE 6 'bubble after' notation suggests that the source state and transition activate together in a reaction.)

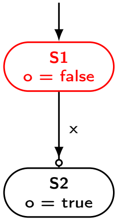
Weak transitions

Simple state machine: o = false until x becomes true, then o = true from the next instant onwards.

Weak preemption: o becomes true the instant after x does.

```
automaton
  state S1 do
    o = false
  until x then S2

  state S2 do
    o = true
end
```



x	F
o	F

(The SCADE 6 ‘bubble after’ notation suggests that the source state and transition activate together in a reaction.)

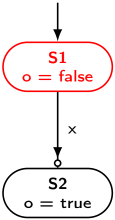
Weak transitions

Simple state machine: o = false until x becomes true, then o = true from the next instant onwards.

Weak preemption: o becomes true the instant after x does.

```
automaton
  state S1 do
    o = false
  until x then S2

  state S2 do
    o = true
end
```



x	F	F
o	F	F

(The SCADE 6 'bubble after' notation suggests that the source state and transition activate together in a reaction.)

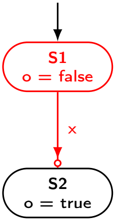
Weak transitions

Simple state machine: o = false until x becomes true, then o = true from the next instant onwards.

Weak preemption: o becomes true the instant after x does.

```
automaton
  state S1 do
    o = false
  until x then S2

  state S2 do
    o = true
end
```



x	F	F	T
o	F	F	F

(The SCADE 6 ‘bubble after’ notation suggests that the source state and transition activate together in a reaction.)

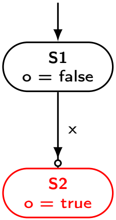
Weak transitions

Simple state machine: o = false until x becomes true, then o = true from the next instant onwards.

Weak preemption: o becomes true the instant after x does.

```
automaton
  state S1 do
    o = false
  until x then S2

  state S2 do
    o = true
end
```



x	F	F	T	F
o	F	F	F	T

(The SCADE 6 'bubble after' notation suggests that the source state and transition activate together in a reaction.)

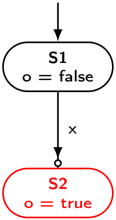
Weak transitions

Simple state machine: o = false until x becomes true, then o = true from the next instant onwards.

Weak preemption: o becomes true the instant after x does.

```
automaton
  state S1 do
    o = false
  until x then S2

  state S2 do
    o = true
end
```



x	F	F	T	F	F
o	F	F	F	T	T

(The SCADE 6 ‘bubble after’ notation suggests that the source state and transition activate together in a reaction.)

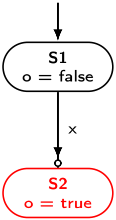
Weak transitions

Simple state machine: o = false until x becomes true, then o = true from the next instant onwards.

Weak preemption: o becomes true the instant after x does.

```
automaton
  state S1 do
    o = false
  until x then S2

  state S2 do
    o = true
end
```



x	F	F	T	F	F	T
o	F	F	F	T	T	T

(The SCADE 6 ‘bubble after’ notation suggests that the source state and transition activate together in a reaction.)

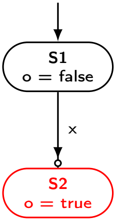
Weak transitions

Simple state machine: o = false until x becomes true, then o = true from the next instant onwards.

Weak preemption: o becomes true the instant after x does.

```
automaton
  state S1 do
    o = false
  until x then S2

  state S2 do
    o = true
end
```



x	F	F	T	F	F	T	...
o	F	F	F	T	T	T	...

(The SCADE 6 ‘bubble after’ notation suggests that the source state and transition activate together in a reaction.)

Up/down counter

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

Up/down counter

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until. . .

First attempt

```
node two_states(min, max : int)
returns (o : int);
let
  automaton
    state Up do
      o = 0 fby (o + 1);
    until (o = max) then Down

    state Down do
      o = 0 fby (o − 1);
    until (o = min) then Up
  end
tel
```

Up/down counter

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

First attempt

```
node two_states(min, max : int)
returns (o : int);
let
  automaton
    state Up do
      o = 0 fby (o + 1);
    until (o = max) then Down

    state Down do
      o = 0 fby (o − 1);
    until (o = min) then Up
  end
tel
```

What's wrong with this implementation?

Up/down counter

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until. . .

First attempt

```
node two_states(min, max : int)
returns (o : int);
let
 automaton
  state Up do
   o = 0 fby (o + 1);
  until (o = max) then Down

  state Down do
   o = 0 fby (o − 1);
  until (o = min) then Up
 end
tel
```

What's wrong with this implementation?

Each state has a separate instance of the fby, which is reset on entry into the state.

Up/down counter

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

Second attempt

```
node two_states(min, max : int)
returns (o : int);
let
  automaton
    state Up do
      o = 0 fby (o + 1);
    until (o = max) continue Down

    state Down do
      o = 0 fby (o − 1);
    until (o = min) continue Up
  end
tel
```

What's wrong with this implementation?

Each state has a separate instance of the fby, which is reset on entry into the state.

Changing ... then S to ... continue S specifies an entry-by-history transition; states are no longer reset on entry...

Up/down counter

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until. . .

Second attempt

```
node two_states(min, max : int)
returns (o : int);
let
  automaton
    state Up do
      o = 0 fby (o + 1);
    until (o = max) continue Down

    state Down do
      o = 0 fby (o − 1);
    until (o = min) continue Up
  end
tel
```

What's wrong with this implementation?

Each state has a separate instance of the fby, which is reset on entry into the state.

Changing ... then S to ... continue S specifies an entry-by-history transition; states are no longer reset on entry. . .

. . . but there are still two counters.

Up/down counter take 2

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

Up/down counter take 2

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

A correct version

```
node two_states(min, max : int)
returns (o : int);
var last_o : int;
let
  last_o = 0 fby o;
  automaton
    state Up do
      o = last_o + 1;
    until (o = max) then Down

    state Down do
      o = last_o - 1;
    until (o = min) then Up
  end
tel
```

Up/down counter take 2

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

A correct version

```
node two_states(min, max : int)
returns (o : int);
var last_o : int;
let
  last_o = 0 fby o;
  automaton
    state Up do
      o = last_o + 1;
    until (o = max) then Down

    state Down do
      o = last_o − 1;
    until (o = min) then Up
  end
tel
```

Here the shared counter is declared outside the state machine.

Up/down counter take 2

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

A correct version

```
node two_states(min, max : int)
returns (o : int);
var last_o : int;
let
  last_o = 0 fby o;
  automaton
    state Up do
      o = last_o + 1;
    until (o = max) then Down

    state Down do
      o = last_o − 1;
    until (o = min) then Up
  end
tel
```

Here the shared counter is declared outside the state machine.

Each state now determines how the shared variable evolves.

Up/down counter take 2

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

With the last operator

```
node two_states(min, max : int)
returns (last o : int = 0);
let
  automaton
    state Up do
      o = last o + 1;
    until (o = max) then Down

    state Down do
      o = last o − 1;
    until (o = min) then Up
  end
tel
```

Here the shared counter is declared outside the state machine.

Each state now determines how the shared variable evolves.

A shared variable can also be declared with last and an initial value, and then accessed with the last operator.

Up/down counter take 2

A two-state automaton that counts up from 0 until it reaches an upper-bound max and then counts down until it reaches a minimum bound min and then counts up until...

With the last operator

```
node two_states(min, max : int)
returns (last o : int = 0);
let
  automaton
    state Up do
      o = last o + 1;
    until (o = max) then Down

    state Down do
      o = last o − 1;
    until (o = min) then Up
  end
tel
```

Here the shared counter is declared outside the state machine.

Each state now determines how the shared variable evolves.

A shared variable can also be declared with last and an initial value, and then accessed with the last operator.

In this case, o need not be defined in every state; its implicit definition is o = last o.

Exercise: programming with state machines

- Add a pause feature to the two states counter.
  When toggle_pause is true the counter is paused or resumed.
  node two_states(min, max : int; toggle_pause : bool) returns (last o : int = 0);

- Reimplement two_states (without toggle_pause) using strong preemption.
  node strong_two_states(min, max : int) returns (last o : int = 0);

- **Without** using automata, implement a “flip/flop switch”
  node bswitch (orig, son, soff : bool) returns (s : bool);
  - Outputs a state `s`, initially set to `orig`.
  - The `on` ‘button’ sets the state to `true` in the next instant.
  - The `off` ‘button’ sets the state to `false` in the next instant.
  - Otherwise the state is unchanged (`assert (not (on and off))`).

- Now reimplement the flip/flop switch with automata and check that both versions had identical behavior.
  node autoswitch (orig, bon, boff : bool) returns (s : bool);

State machines: summary

- Lustre + State machines
  - Natural definitions of parallel composition and refinement (hierarchy)
  - Easily mix control- and data- dominated behaviours.
  - Strong and weak preemption, entry-by-history, shared variables
- Limited actions per cycle:
  - More restrictive than other formalisms (Statecharts, Simulink/Stateflow)
  - Easier to reason about, analyze, and compile.
- Useful for mode-driven behaviours, e.g., different control equations at take-off, cruising, and landing.
- Useful for sequential logic, e.g., sequencing operations, responding to commands and exceptional conditions.
- Typical use-case: aircraft user interfaces [Colaço, Pagano, and Pouzet (2017): Scade 6: A Formal Language for Embedded Critical Software Development]

Further reading

- [Bourke and Pouzet (2013): Zélus: A Synchronous Language with ODEs]
- [Biernacki, Colaço, Hamon, and Pouzet (2008): Clock-directed modular code generation for synchronous data-flow languages]

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.
- Bourke, T. and M. Pouzet (Apr. 2013). "Zélus: A Synchronous Language with ODEs". In: *Proc. 16th Int. Conf. on Hybrid Systems: Computation and Control (HSCC 2013)*. Ed. by C. Belta and F. Ivancic. Philadelphia, USA: ACM Press, pp. 113–118.
- Caspi, P., N. Halbwachs, F. Maraninchi, L. Morel, and P. Raymond (2014). *Arrays in Lustre*. Slides for ER02 research school: "Synchronous Approaches for Embedded Systems".

References II

- 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.
- 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. 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.

References III

- Harel, D. (June 1987). "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3, pp. 231–274.
- Heptagon Developers (Apr. 2017). *Heptagon/BZR manual*.
- Maraninchi, F. and N. Halbwachs (Sept. 1996). "Compiling Argos into Boolean equations". In: *Proc. 4th Int. Symp. Formal Techniques in Real-Time and Fault-Tolerance Systems (FTRTFT '96)*. Ed. by B. Jonsson and J. Parrow. Vol. 1135. LNCS. Uppsala, Sweden: Springer, pp. 72–89.
- Maraninchi, F. and Y. Rémond (2001). "Argos: an automaton-based synchronous language". In: *Computer Languages* 27.1–3, pp. 61–92.
- — (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.
- Morel, L. (Mar. 2007). "Array Iterators in Lustre: From a Language Extension to Its Exploitation in Validation". In: *EURASIP Journal of Embedded Systems*. 1. Springer, Article 059130.

References IV

- Pouzet, M. (Apr. 2006). *Lucid Synchrone, v. 3. Tutorial and reference manual*. Université Paris-Sud.
- Rocheteau, F. (June 1992). "Extension du langage LUSTRE et application à la conception de circuits: le langage LUSTRE-V4 et le système POLLUX". PhD thesis. Grenoble: Institut National Polytechnique de Grenoble (INPG).