Synchronous Programming of Reactive Systems

Nicolas Halbwachs

Verimag/CNRS
Grenoble

Outline

Introduction

Reactive Systems

- Permanent reaction to an environment that cannot wait
- Embedded systems

  e.g., transportation, industrial control

Reactive Systems

Specific features

- deterministic
- concurrent (logical $\neq$ physical)
- safety critical

Logical concurrency

ex. A digital watch:

- time keeper
- alarm
- stopwatch
- display manager
- button handler

Design these modules separately, compose them concurrently.

Usual (asynchronous) languages for concurrency don't work

Example: Every 60 seconds, emit a signal MINUTE
An attempt in ADA style:

```
task A:   loop
             delay 60; B.MINUTE!
          end
```

- Rendez-vous (symmetric communication) doesn't work.
- Non-deterministic scheduling (asynchronous interleaving) doesn't work.
- No broadcasting

How are reactive systems commonly implemented? (1/3)

Simple implementation (event driven)

```
< Intialize Memory >
foreach input_event do
    < Compute Outputs >
    < Update Memory >
end
```


How are reactive systems commonly implemented? (2/3)

Even simpler implementation (periodic sampling)

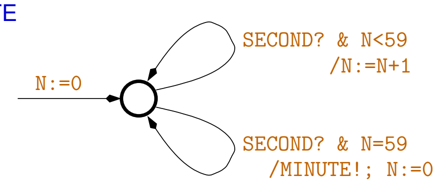
```
< Intialize Memory >
foreach period do
    < Read Inputs >
    < Compute Outputs >
    < Update Memory >
end
```


How are reactive systems commonly implemented? (3/3)

∼ interpreted automaton
a loop iteration = a transition = a logical instant

Our example in Esterel:

```
every 60 SECOND do
    emit MINUTE
end
```



"Real-time" correctness condition
max transition time $<$ min environment delay

Synchronous programming
= high level, structured, modular
description of interpreted automata

concurrency = synchronous product

```
[
    every 60 SECOND do emit MINUTE end
||
    every 60 MINUTE do emit HOUR end
]
```


Another point of view: time, concurrency, and compositionality

$\Delta(f(x))$?

depends on implementation of f, on the target machine, and generally on x

Abstraction: $\Delta(f(x)) = \delta$

Compositionality: $f(x) = g(h(x))$

$$\Delta_f = \Delta_g + \Delta_h \qquad \delta = \delta + \delta$$

Another point of view: time, concurrency, and compositionality

$\Delta(f(x))$?

depends on implementation of f, on the target machine, and generally on x

Abstraction: $\Delta(f(x)) = \delta$

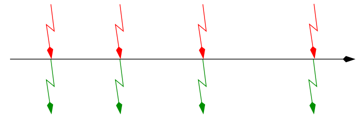
Compositionality: $f(x) = g(h(x))$

$$\Delta_f = \Delta_g + \Delta_h \qquad \delta = \delta + \delta$$

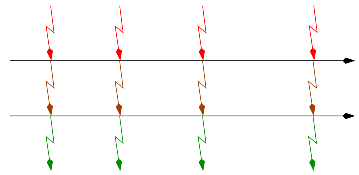
Two solutions:
$\delta = 0$ (synchrony), $\delta = ?$ (asynchrony)

Abstract synchronous behavior

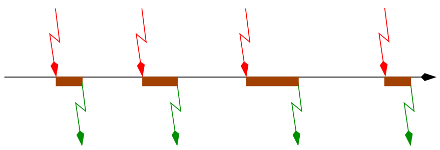
sequence of reactions to input events, to which all processes take part:



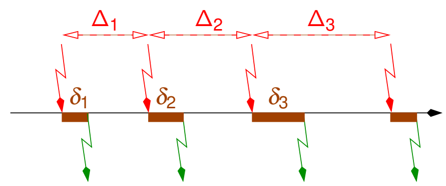
Composition of behaviors:



Concrete behavior



Concrete behavior



Valid abstraction as long as $\delta_i < \Delta_i$

What's new?

Classical in synchronous circuits

- synchronous communicating Mealy machines
- dynamic Boolean equations
- gate and latch networks

What's new?

Classical in synchronous circuits

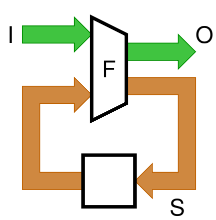
- synchronous communicating Mealy machines
- dynamic Boolean equations
- gate and latch networks

Classical in control engineering

data-flow synchronous formalisms

- differential or finite difference equations
- block-diagrams, analog networks

Connexion with synchronous circuits (1/2)



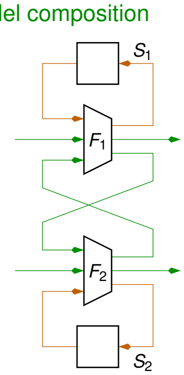
$$(O_n, S_n) = F(I_n, S_{n-1})$$

$\longrightarrow$ Data-flow languages (Lustre/Scade, Signal/Sildex)

In Lustre: (O,S) = F(I, pre(S))

Connexion with synchronous circuits (2/2)

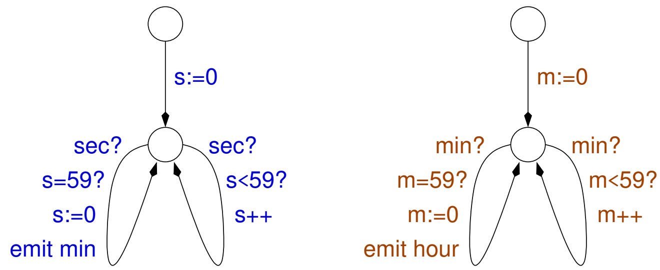
Parallel composition



(S1,L1,M1) = F1(E1,L2,pre(M1))
(S2,L2,M2) = F2(E2,L1,pre(M2))

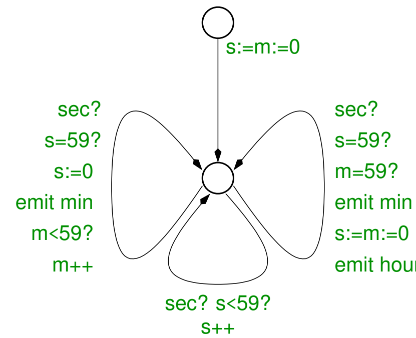
Deterministic! (provided there is no combinational loop)

Connexion with synchronous automata (1/3)



Connexion with synchronous automata (2/3)

Synchronous product of automata



Connexion with synchronous automata (3/3)

$\longrightarrow$ Imperative Languages (Esterel, Synccharts)

In Esterel:

```
every 60 sec do emit min end
||
every 60 min do emit hour end
```


Synchronous Languages

Imperative
- StateCharts
- Esterel
- Argos, SyncChart

Declarative
- Lustre, Signal

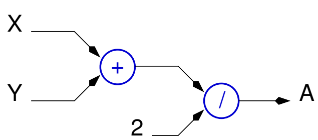
Industrial use

- Avionics, aerospace:
  Airbus, Honeywell, Eurocopter, Astrium (Lustre/Scade)
  Dassault (Esterel)
  Hispano-Suiza (Signal)
- Nuclear plants:
  Rolls-Royce CN, EDF (Lustre)
- CAD:
  Cadence, Synopsys, TI (Esterel)
- Telecom:
  TI (Esterel)
- Many more...

The Data-Flow Language Lustre

The data-flow approach (1/3)

Classical in control theory (equations, data-flow networks) and circuits (equations, gate networks)



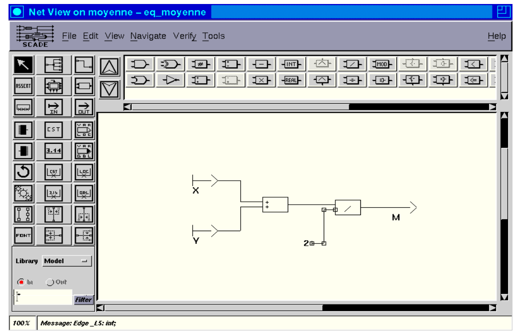
node Average(X,Y: int)
returns (A: int);
let
A = (X+Y)/2;
tel

Synchronous interpretation: time = $\mathbb{N}$

$$\forall n \in \mathbb{N}, A_n = (X_n + Y_n)/2$$

The data-flow approach (2/3)

Lustre (textual) and Scade (graphical)



The data-flow approach (3/3)

Other solution

node Average (X,Y: int) returns (A: int);
var S: int; ← auxiliary variable
let
A = S/2; ← system of equations
S = X+Y; (meaningless order)
tel

Equations and flows

System of equations

- One definition for each output/local variable
- Meaningless order
- Substitution principle (referential transparency)

Equations and flows

System of equations

- One definition for each output/local variable
- Meaningless order
- Substitution principle (referential transparency)

Flows

- Each variable, or constant, or expression, represents an infinite sequence of values

$$X = x_0, x_1, \ldots, x_n, \ldots$$

- x_n is the value of X at the n-th cycle of the program

The combinational part of the language

Base types: bool, int, real

Constants

- 2 = 2, 2, 2, . . .
- true = true, true, true, . . .

Pointwise operators

- standard arithmetic and logic operators

$$X+Y = x_0 + y_0 \;,\; x_1 + y_1 \;,\; \ldots$$

A Boolean example

node Nand (X, Y: bool) returns (nand: bool);
let
nand = not (X and Y);
tel

Execution:

X	true	true	false	true	true	. . .
Y	false	true	false	false	true	. . .
nand	true	false	true	true	false	. . .

Pointwise operators: the conditional operator

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

Execution:

A	1	10	8	25	12	...
B	5	8	8	15	17	...
max	5	10	8	25	17	...

Temporal operators

- "pre" (previous) operator

  One step delay:

X	x_0	x_1	x_2	x_3	x_4	...
pre(X)	*nil*	x_0	x_1	x_2	x_3	...

- "->" (followed-by) operator

  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	...

Formal semantics

- pointwise operators

$$(\mathrm{op}(X, Y, \ldots, Z))_i = op(X_i, Y_i, \ldots, Z_i)$$

- temporal operators

$$(\mathrm{pre}(X))_i = \begin{cases} nil & \text{if } i = 0 \\ X_{i-1} & \text{otherwise} \end{cases}$$

$$(X \texttt{ -> } Y)_i = \begin{cases} X_0 & \text{if } i = 0 \\ Y_i & \text{otherwise} \end{cases}$$

Simple examples

Rising edge of a Boolean flow

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


Simple examples (cont.)

Min and Max values 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 definitions
```


Min and Max values of a sequence (cont.)

Other writing (definitions of pairs):

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

Correct recursive definitions

Example

alt = false -> not pre(alt);

The sequence of values can be computed step-by-step:

- alt_0 = $false_0$ = *false*
- alt_1 = $(not\ pre(alt))_1$ = not alt_0 = *true*
- ...

Correct recursive definitions

Example

alt = false -> not pre(alt);

The sequence of values can be computed step-by-step:

- alt_0 = $false_0$ = *false*
- alt_1 = $(not\ pre(alt))_1$ = not alt_0 = *true*
- ...

Other example

nat = 0 -> pre(nat) + 1;

Incorrect recursive definitions

Example

X = 1/(2-X);

- Unique solution (X=1) but cannot be obtained constructively.
- Any dependence loop (not cut by a "pre") is rejected by the compiler ("causality error"), even:

  X = if C then Y else Z;
  Y = if C then W else X;

Double initialization

Define P ≡ *F, F, T, F, T, F,T, F,...*

Double initialization

Define P ≡ *F, F, T, F, T, F,T, F,...*

P = false -> pre(Q);
Q = false -> not pre(Q);

Warning: X -> Y -> Z = X -> Z

Double initialization

Define P ≡ *F, F, T, F, T, F,T, F,...*

P = false -> pre(Q);
Q = false -> not pre(Q);

Warning: X -> Y -> Z = X -> Z

Define fib ≡ 1, 1, 2, 3, 5, 8, 13, ...

Double initialization

Define P ≡ *F, F, T, F, T, F,T, F,...*

P = false -> pre(Q);
Q = false -> not pre(Q);

Warning: X -> Y -> Z = X -> Z

Define fib ≡ 1, 1, 2, 3, 5, 8, 13, ...

fib = 1 -> pre(f);
f = 1 ->(fib + pre(fib));

or (substitution)

fib = 1 -> pre(1 ->(fib + pre(fib)));

Counter

Write a node
node Counter (B: bool) returns (count: int);
where count is the number of *true* occurrences of B

Counter

Write a node

node Counter (B: bool) returns (count: int);

where count is the number of *true* occurrences of B

```
node Counter (B: bool) returns (count: int);
let
    count = 0 -> if B  then pre(count) + 1
                       else pre(count);
tel
```


Counter

Write a node

node Counter (B: bool) returns (count: int);

where count is the number of *true* occurrences of B

```
node Counter (B: bool) returns (count: int);
let
    count = 0 -> if B  then pre(count) + 1
                       else pre(count);
tel
```

misses possible initial occurrence of B

Counter (cont.)

Better solution:

```
node Counter (B: bool) returns (count: int);
let
    count = if B then (1 -> pre(count) + 1)
                 else (0 -> pre(count));
tel
```


Exercise: Counter with reset

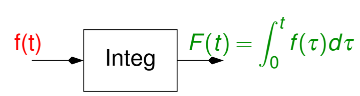
Write a node

```
node RCounter (B, reset: bool)
    returns (count: int);
```

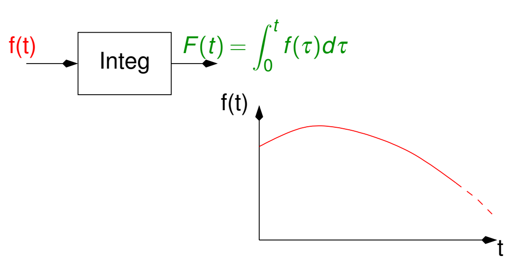
which does the same, but is reset to 0 when "reset" is true

Exercise: Counter with reset (cont.)

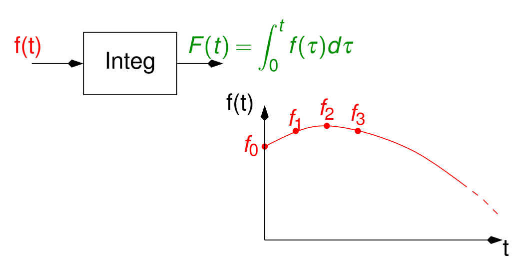
Another example: an integrator



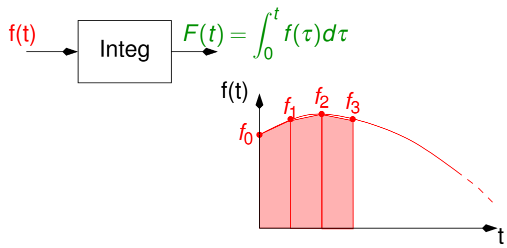
Another example: an integrator



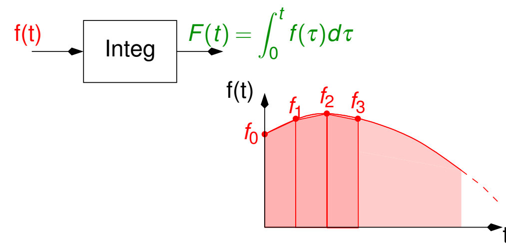
Another example: an integrator



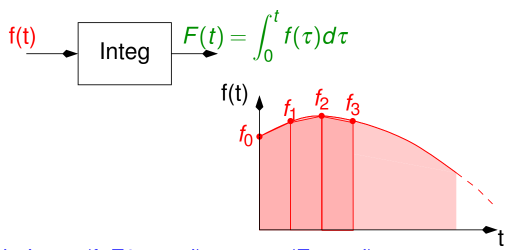
Another example: an integrator



Another example: an integrator



Another example: an integrator




```
node Integ (f, F0 : real) returns (F : real);
let
   F = F0 -> pre(F) + (f + pre(f))/2.0 ;

tel
```


Program structure

Any defined node can be re-used, as a basic operator.

Called as a function:

```
nb_seconds =
     RCounter( second ,
                  second and (false -> pre(nb_seconds = 59) ));
```


Program structure

Any defined node can be re-used, as a basic operator.

Called as a function:

```
nb_seconds =
     RCounter( second ,
                  second and (false -> pre(nb_seconds = 59) ));
```

Calling a node with several outputs:

```
node AvMinMax (X: int) returns (A: int);
var min, max : int;
let
     A = Average(min, max);
     (min, max) = MinMax(X);
tel
```


Other example: Switches

Write a node TwoStates receiving 3 Boolean inputs init, set, and reset, and behaving like a switch or a flip-flop:

Its boolean output state

- is initially equal to init,
- is set to true when set is true,
- is reset to false when reset is true,
- keeps its previous value otherwise.

Switches (cont.)

First solution

```
node TwoStates(init, set, reset: bool)
returns (state: bool);
let
     state = init -> if set then true
                     else if reset then false
                     else pre(state);
tel
```


Switches (cont.)

First solution

```
node TwoStates(init, set, reset: bool)
returns (state: bool);
let
     state = init -> if set then true
                     else if reset then false
                     else pre(state);
tel
```

Problem: what happens with the call
   TwoStates(false, change, change)?

Switches (cont.)

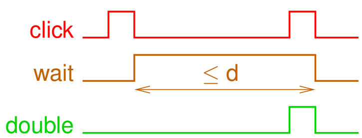
Can you fix this problem?

A more complex example: mouse click

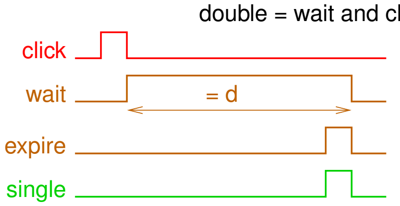
Write a node Mouse, receiving an event click (a boolean input) and computing two events (boolean outputs) single and double as follows:

- single occurs whenever a click occurs alone, i.e., is not followed by another click within d cycles (d is a constant integer parameter)
- double occurs when a second click follows within the delay d.

Mouse click (cont.)

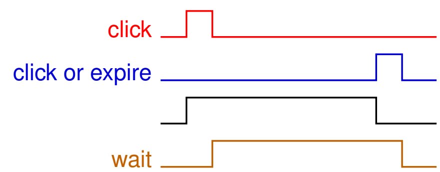


double = wait and click



single = expire and not click

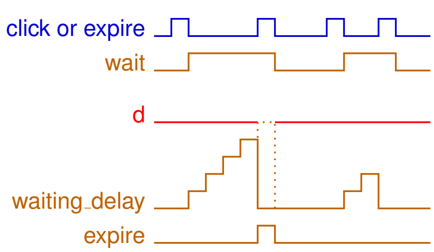
Mouse click (cont.)



wait = false -> pre(TwoStates(click,click,click or expire))

=TwoStates (false,pre(click),pre(click or expire))

Mouse click (cont.)



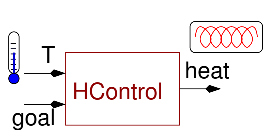
waiting_delay = RCounter(click or expire, wait)
expire = false -> pre(waiting_delay) = d-1

Mouse click: complete program

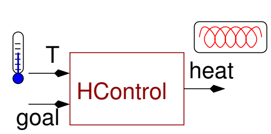
```
node Mouse (d: int; click: bool)
    returns (single, double: bool);
var wait, expire: bool; waiting_delay: int;
let  single = expire and not click;
     double = wait and click;
     expire = false -> pre(waiting_delay)=d-1;
     wait = false ->
               pre(TwoStates(click, click, click or expire));
     waiting_delay = Counter(click or expire, wait);
tel
```

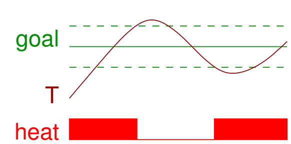
DEMO

A complete example: Heating control

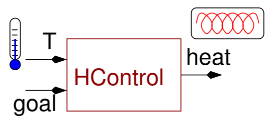


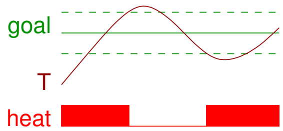
A complete example: Heating control





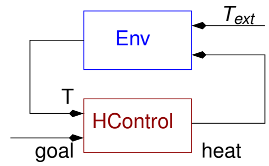
A complete example: Heating control






```
node HControl (T, goal : real) returns (heat: bool);
let
    heat = true -> if (T > goal + 1.) then false
                   else if (T < goal -1.) then true
                   else pre(heat);
tel
```


Heating control: simulating the environment



When the heater is on :

$$T' = \alpha(h - T)$$

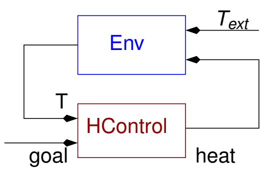
When it is off :

$$T' = \beta(T_{ext} - T)$$

```
node Env (heat: bool; ExtT: real) returns (T: real);
var Tprime : real;
let
  Tprime = if heat then alpha * (h-T) else beta * (ExtT-T);
  T = Integ(Tprime, ExtT);
tel
```

COMPIL

Heating control: simulating the environment



When the heater is on :

$$T' = \alpha(h - T)$$

When it is off :

$$T' = \beta(T_{ext} - T)$$

```
node Env (heat: bool; ExtT: real) returns (T: real);
var Tprime : real;
let
  Tprime = if heat then alpha * (h-T) else beta * (ExtT-T);
  T = Integ(Tprime, ExtT);
  T = Integ(0.0 -> pre(Tprime), ExtT);
tel
```

COMPIL

Heating control: Simulation program

```
node Simul (goal, ExtT: real)
    returns (heat: bool; T: real);
let
    heat = HControl(T, goal);
    T = Env(heat, ExtT);
tel
```

DEMO

Clocks

Sampling: the “when” operator

Goal: define a flow “slower” that inputs

X	4	1	-3	0	2	7	8	3	12
C	F	T	F	F	T	T	F	T	F
X when C		1			2	7		3	

When C is false, X when C does not exist

One may only operate on flows with the same clock:
X + (X when C) is forbidden

Clocks (cont.)

Projection: the “current” operator

Goal: Coming back to a “faster” clock

X	4	1	-3	0	2	7	8	3	12
C	F	T	F	F	T	T	F	T	F
X when C		1			2	7		3	
current(X when C)	nil	1	1	1	2	7	7	3	3

memory!

Clocks: typical use

X	4	1	-3	0	2	7	8	3
C	F	T	F	F	T	T	F	T
current(X when C)	nil	1	1	1	2	7	7	3

Clocks: Initialization problem

Solution 1: clock initially true

X	4	1	-3	0	2	7	8	3
C	F	T	F	F	T	T	F	T
Ci = (true -> C)	T	T	F	F	T	T	F	T
current(X when Ci)	4	1	1	1	2	7	7	3

Clocks: Initialization problem

Solution 2: Forcing an initial value

```
E = if C then current(X when C)
         else (default -> pre(E));
```


Clocks: Initialization problem

Solution 2: Forcing an initial value

```
E = if C then current(X when C)
         else (default -> pre(E));
```

or (to save a memory)

```
X1 = (if C then X else default) -> X;
E = current(X1 when C);
```


Nodes and clocks

A node works at the rate of its actual input parameters (data-flow behavior)

C	F	T	F	T	F	F	T	T
true when C		T		T			T	T
Counter(true when C)		1		2			3	4

sampling of inputs $\neq$ sampling of outputs

Counter(true)	1	2	3	4	5	6	7	8
Counter(true) when C		2		4			7	8

Assertions

assert ⟨ bool_exp ⟩ ;

allows the programmer to write general assumptions (about the program environment).

Examples:

```
assert not (R and S);
assert true -> (U - pre(U)) <= 5;
```


Assertions (cont.)

These assertions are taken into account

- by the simulator
- by the compiler (under some options)
- by verification and testing tools

The story of Lustre

- 1984: first design (Caspi, Halbwachs - IMAG)
- 1987: first academic compiler (Plaice - IMAG)
- 1989: first industrial environment, SAGA (Verilog, Schneider-Electric)
- 1995: SAGA becomes SCADE (Verilog, Schneider-Electric, Airbus)
- 2001: SCADE is bought by Esterel-Technologies see www.esterel-technologies.com/v2/
- 2006: SCADE V6 (automata, arrays, . . .)

The Imperative Language

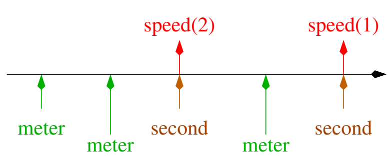
Esterel

Introductory example: a speedometer

- receives signals Second and Meter
- each Second emits a signal Speed carrying the number of Meters received during the last Second

```
module Speedometer:
input Second, Meter ; output Speed : integer in
  loop   var Distance := 0 : integer in
           abort
             every Meter do Distance := Distance+1
             end every
           when Second ;
           emit Speed(Distance)
         end var
  end loop
end module
```


- An Esterel process communicates with its environment by means of (pure or valued) signals.
- Its behavior is a sequence of reactions, each of which triggered by input signals
  Ex.: a behavior of the speedometer



- In a reaction, a signal is either present or absent (instantaneous broadcast)


```
module Speedometer:
input second, meter ;
output speed: integer in
  loop
    var Distance := 0 : integer in
      abort
        every meter do
          Distance :=
              Distance+1
        end every
      when second ;
      emit speed(Distance);
    end var
  end loop
end module
```

Distance
0

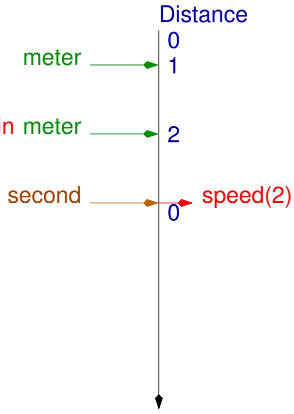

```
module Speedometer:
input second, meter ;
output speed: integer in
  loop
    var Distance := 0 : integer in
      abort
        every meter do
          Distance :=
              Distance+1
        end every
      when second ;
      emit speed(Distance);
    end var
  end loop
end module
```

Distance
0
meter → 1

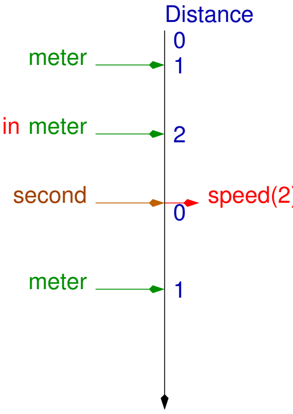

```
module Speedometer:
input second, meter ;
output speed: integer in
  loop
    var Distance := 0 : integer in
      abort
        every meter do
          Distance :=
              Distance+1
        end every
      when second ;
      emit speed(Distance);
    end var
  end loop
end module
```

Distance
0
meter → 1
meter → 2

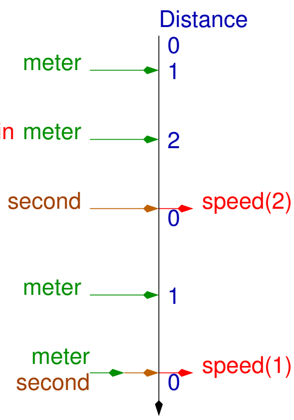
```
module Speedometer:
input second, meter ;
output speed: integer in
  loop
    var Distance := 0 : integer in meter
      abort
        every meter do
          Distance :=
              Distance+1
        end every
      when second ;
      emit speed(Distance);
    end var
  end loop
end module
```




```
module Speedometer:
input second, meter ;
output speed: integer in
  loop
    var Distance := 0 : integer in meter
      abort
        every meter do
          Distance :=
              Distance+1
        end every
      when second ;
      emit speed(Distance);
    end var
  end loop
end module
```




```
module Speedometer:
input second, meter ;
output speed: integer in
  loop
    var Distance := 0 : integer in meter
      abort
        every meter do
          Distance :=
              Distance+1
        end every
      when second ;
      emit speed(Distance);
    end var
  end loop
end module
```




```
module Speedometer:
input Second, Meter ;
output Speed : integer in
 loop
  var Distance := 0 : integer in
   abort
    every Meter do
     Distance := Distance+1
    end every
   when Second ;
   emit Speed(Distance)
  end var
 end loop
end module
```

loop Stat end
abort Stat when Occ
every Occ do Stat end

```
  = await Occ ;
    loop
       abort
          Stat ; halt
       when Occ
    end
```

await Occ
  = abort halt when Occ

Signals , Events, and occurrences

- Event = set of present signals
- A reaction consists of adding signals to an input event.
- tick is a special signal which belongs to all events
- If S is a valued signal, ?S refers to the value carried by its last occurrence

Signals , Events, and occurrences

- Occurrences:
  In statements like await S, abort ... when S, the considered occurrence of S is in the strict future.
  but you can write also await immediate S, ...
- An occurrence may also consist of a number of signal occurrences:

  every 60 SEC do emit MIN end

Kernel language

nothing, halt
v := exp
stat ; stat
if exp then stat else stat
loop stat end
trap id in stat
exit id

var var_decls in stat
signal signal_decls in stat
[stat || stat]
emit S , emit S(exp)
present S then stat1 else stat2
abort stat when S

Some derived statements

```
await S          (= abort halt when S)
weak abort stat when S
                 (= trap T in [ stat; exit T || await S; exit T] end )
pause            (= await tick)
```


Some derived statements

```
abort stat1 when S timeout stat2 end
                   =  trap T in
                         abort stat1; exit T when S;
                         stat2
                      end trap
```


Some derived statements

```
abort stat1 when S timeout stat2 end
                   =  trap T in
                         abort stat1; exit T when S;
                         stat2
                      end trap

every Occ do stat =   await Occ ;
                      loop
                         abort
                            Stat ; halt
                         when Occ ;
                      end loop
```


Example 1: Parallel composition and instantaneous broadcast

```
input Second;
output Minute, Hour;
[
   every 60 Second do emit Minute end
||
   every 60 Minute do emit Hour end
]
```

Exercise: ABRO

emit O as soon as both A and B have occurred. Restart on any occurrence of R.

Structure nesting and priorities

Example 2: Mouse Click

- receives pure signals click and hsec.
- emits double whenever two clicks happen within d hsec, and single whenever a click is not followed by a second click within that delay.

Structure nesting and priorities

```
module mouse:
  constant d: integer;
  input click, hsec; output single, double;




end
```


Structure nesting and priorities

```
module mouse:
  constant d: integer;
  input click, hsec; output single, double;

    await click;



end
```


Structure nesting and priorities

```
module mouse:
  constant d: integer;
  input click, hsec; output single, double;

    await click;
    abort
      await click; emit double
    when d hsec

end
```


Structure nesting and priorities

```
module mouse:
  constant d: integer;
  input click, hsec; output single, double;

    await click;
    abort
      await click; emit double
    when d hsec
    timeout emit single end

end
```


Structure nesting and priorities

```
module mouse:
  constant d: integer;
  input click, hsec; output single, double;
  loop
    await click;
    abort
      await click; emit double
    when d hsec
    timeout emit single end
  end loop
end
```


Structure nesting and priorities

```
module mouse:
  constant d: integer;
  input click, hsec; output single, double;
  loop
    await click;
    abort
      await click; emit double
    when d hsec
    timeout emit single end
  end loop
end
```

slightly wrong when simultaneous “click” and “d hsec”

Structure nesting and priorities

Correct solution:

```
module mouse:
  constant d: integer;
  input click, hsec; output single, double;
  loop
    await click;
    abort
      await d hsec; emit single
    when click
    timeout emit double end
  end loop
end
```


Structure nesting and priorities

```
loop
  await click;
  abort
    await click;
    emit double
  when d hsec
  timeout emit single end
end loop
```

```
loop
  await click;
  abort
    await d hsec;
    emit single
  when click
  timeout emit double end
end loop
```

DEMO

Exercise: Reflex game

- The player puts a coin in the machine
- After a random delay, the machine switches on the go lamp
  The player should press the stop button as soon as possible
- Then the machine displays the time (in ms.) elapsed between go and stop. The go is switched off, the game_over lamp is switched on, and a new game can start
- Exception cases:
  - the player presses the stop before go (cheating! ring the bell and end the game)
  - the player does not press stop within limit_time ms. after the go lamp is on (abandon, end the game)
- Initially, only the game_over in on, the machine displays 0.

Reflex game: declarations

```
module REFLEX_GAME :
constant limit_time : integer;
function RANDOM() : integer;
input MS, COIN, STOP;
output    DISPLAY(integer),
          GO_ON, GO_OFF,
          GameOver_ON, GameOver_OFF,
          RingBell;
```


Reflex game: general structure

```
overall initializations
every COIN do % loop over each game
  game initialization
  trap END_GAME in % exception handling
    game
  end % trap END_GAME
end % every COIN
end of a game
```


Reflex game: initializations

- Overall initializations:
  ```
  emit DISPLAY(0);
  emit GO_OFF;
  emit GameOver_ON;
  ```


Reflex game: initializations

- Overall initializations:
  ```
  emit DISPLAY(0);
  emit GO_OFF;
  emit GameOver_ON;
  ```
- Game initializations
  ```
  emit DISPLAY(0);
  emit GO_OFF;
  emit GameOver_OFF;
  ```


Reflex game: the game (1) [Exercise]

```
% Phase 1:
% wait for a random delay, and switch on the GO lamp
% if STOP is pushed while waiting,
%       ring bell and end the game
```

Reflex game: the game (2) [Exercise]

% Phase 2: count the time until STOP
% if limit_time is reached, end the game

Reflex game: end of a game

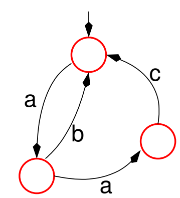
emit GO_OFF;
emit Game_Over_ON;

The story of Esterel

- developped since early 80th at CMA/ENSMP and Inria [G. Berry & co.]
- now equipped with a graphical syntax: SyncCharts [Ch. Andre, 1996]
- commercialized by Esterel-Technologies until 2009
- several sources, many users
- see www.esterel.org

SyncCharts: a graphical language based on Esterel

- Automata are very useful for describing control
- The best way for describing an automaton is by drawing it



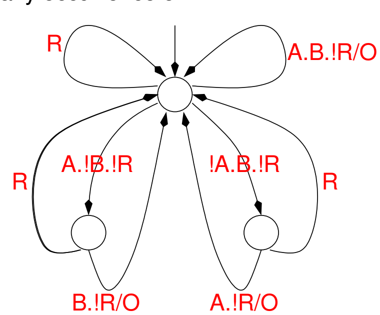
(not always easy to specify in Esterel)

Graphical, automata-based language: Needs

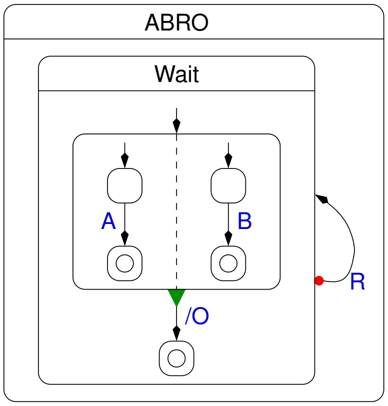
- Structure (cf. Statecharts: parallel and hierarchical composition)
- Avoid text (texts and drawings don't merge so well)
- Small number of graphical primitives
- (Of course!) Precise semantics

Example: ABRO as an automaton

emit O as soon as both A and B have occurred.
Restart on any occurrence of R.



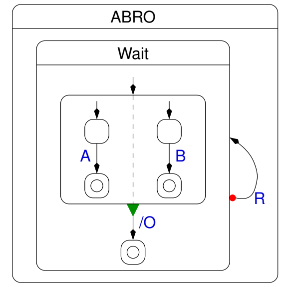
ABRO in SyncCharts



ABRO in SyncCharts

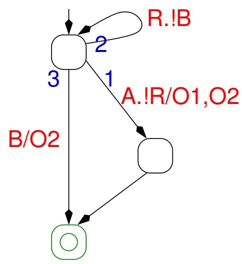
Advantages

- Better structure
- Each signal appears once
- Waiting also for C doesn't complexify too much (linear increase)



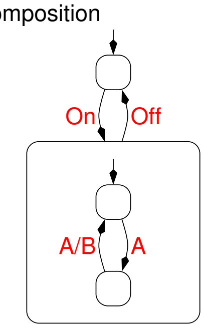
SyncCharts: Basic constructs (1)

Automata with
(input events)/(output signals)
on the transitions,
possibly final states,
possible priorities on transitions.



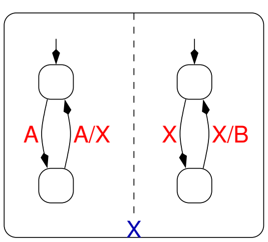
SyncCharts: Basic constructs (2)

Hierarchical composition



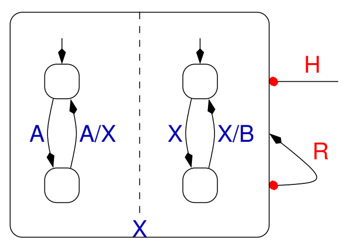
SyncCharts: Basic constructs (3)

Parallel composition, local signal



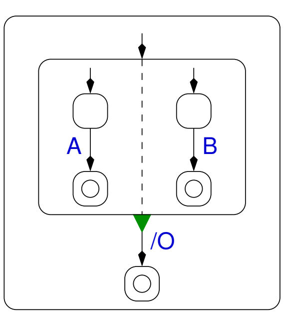
SyncCharts: Basic constructs (4)

Inhibition and inhibiting transitions

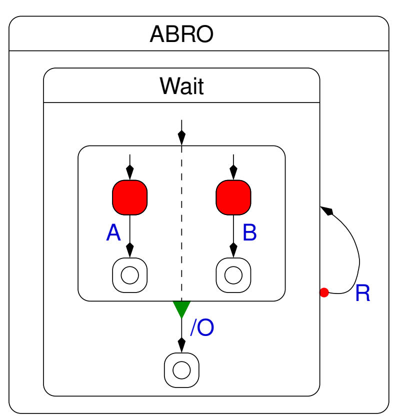


SyncCharts: Basic constructs (5)

Transitions on termination

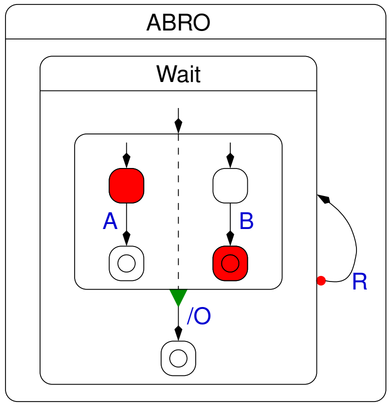


ABRO simulation



Initial

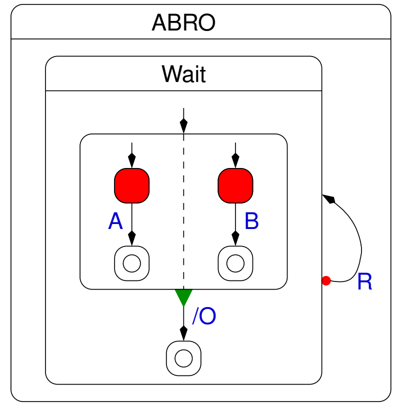
ABRO simulation



Present signals

B

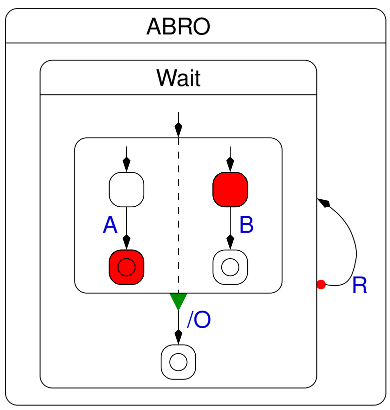
ABRO simulation



Present signals

A, R

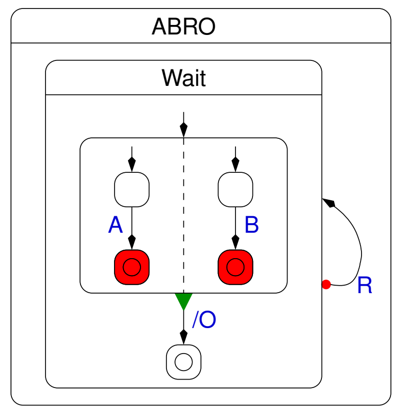
ABRO simulation



Present signals

A

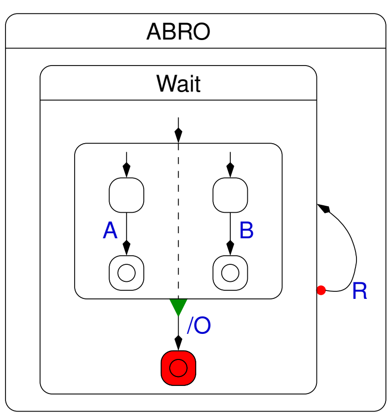
ABRO simulation



Present signals

B

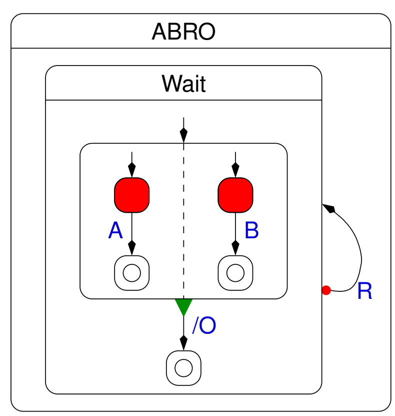
ABRO simulation



Present signals

B
O

ABRO simulation



Present signals

R

Advantages of SyncCharts over Esterel

- Graphical (people sometimes like it ?!)
- Easy automata description
- Basically, same difference as between automata and regular expressions
- Nesting of interrupts/preemption/exceptions more readable than with textual nesting

Compilation of Synchronous Languages

14 Static semantics
- Causality analysis

15 Sequential code generation

14 Static semantics
- Causality analysis

Static semantics

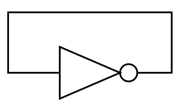
Causality errors

Some programs don't make sense
($\sim$ combinational loops in circuits)

In Lustre

x = not x ;

In circuits



In Esterel

module P1: output x ;
present x else emit x
end present
end module

no behavior

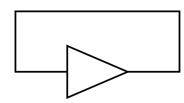
Causality errors (1/3)

Other bad case:

In Lustre

x = x ;

In circuits



In Esterel

module P1:
output x ;
present x
then emit x
end present
end module

many behaviors (nondetermism?)

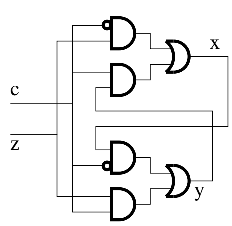
Causality errors (2/3)

but some loops do make sense !

In Lustre

x = if c then y else z ;
y = if c then z else x ;

In circuits



In Esterel

```
module P1:
input c, z ; output x, y ;
   present c
   [   present y then emit x end
   ||  present z then emit y end
   ]
   else
   [   present z then emit x end
   ||  present x then emit y end
   ]
   end present
end module
```


Causality errors (3/3)

- Source of all the problems with the semantics of Statecharts and Sequential Function Charts (Grafcet)
- Where does the problem come from ?
  The result of a reaction is a fixpoint of a function which is not necessarily increasing (because of the negation, or the reaction to absence)

Causality analysis (1/4)

Solutions :
- forbid loops (Lustre)
- forbid instantaneous reaction to absence
  (SL [Boussinot], ReactiveML [Mandel])

In pure Esterel, or in Boolean Lustre, the problem boils down to determining whether a system of Boolean equations has one and only one solution.
- one and only one behavior in classical logic
- one and only one behavior in constructive logic (Esterel V5)

Causality analysis (2/4)

A strange Lustre program:

x = x;
y = x and not y;

only one behavior in classical logic
no behavior in constructive logic!

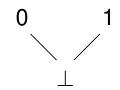
Causality analysis (3/4)

Computing in constructive logic:

- Use Scott's Boolean domain with

$$0 \wedge \bot = \bot \wedge 0 = 0$$

$$1 \vee \bot = \bot \vee 1 = 1$$



- or use dual rail encoding :

$$x_0 x_1 \quad \text{with} \quad \begin{cases} x_0 = 1 & \text{iff } x \text{ surely } 0 \\ x_1 = 1 & \text{iff } x \text{ surely } 1 \end{cases}$$

Causality analysis (4/4)

Example:

x = x ;
y = x and not y ;

Causality analysis (4/4)

Example:

x = x ;	$\bot$
y = x and not y ;	$\bot$

Causality analysis (4/4)

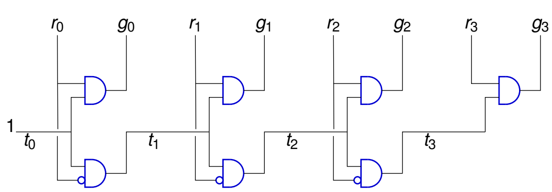
Example:

x = x ;	$\bot$	$\bot$
y = x and not y ;	$\bot$	$\bot$

Example : McMillan/DeSimone's bus arbiter

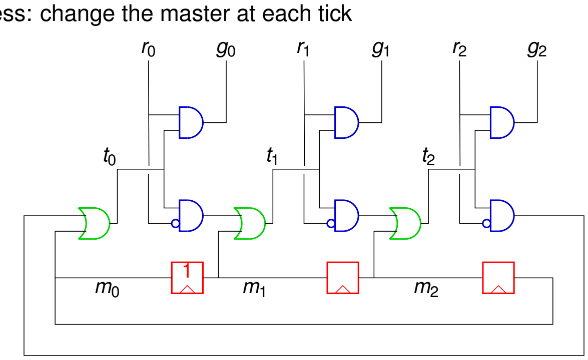
n units connected to a bus. At each clock tick, some of them can ask for bus access for this tick. At most one may be granted, and the allocation should be fair.

Basic idea: travelling token:



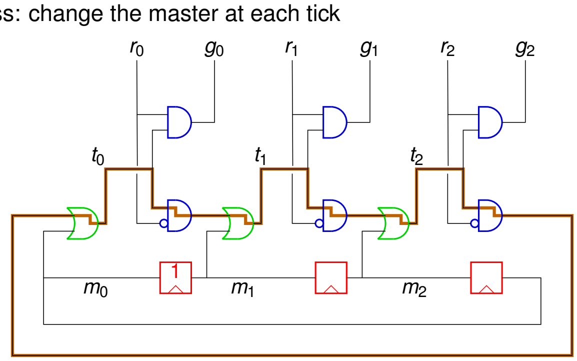
Example : McMillan/DeSimone's bus arbiter

Fairness: change the master at each tick



Example : McMillan/DeSimone's bus arbiter

Fairness: change the master at each tick



Combinational loop!

Example : McMillan/DeSimone's bus arbiter

$$t_0 = m_0 + t_2.\overline{r_2} \quad , \quad t_1 = m_1 + t_0.\overline{r_0} \quad , \quad t_2 = m_2 + t_1.\overline{r_1}$$

but. . .

- if $m_0 = 1$, $t_0 = 1$,

$$t_1 = m_1 + \overline{r_0}, \quad t_2 = m_2 + (m_1 + \overline{r_0}).\overline{r_1}$$

- if $m_1 = 1$, $t_1 = 1$,

$$t_2 = m_2 + \overline{r_1}, \quad t_0 = m_0 + (m_2 + \overline{r_2}).\overline{r_0}$$

- if $m_2 = 1$, $t_2 = 1$,

$$t_0 = m_0 + \overline{r_2}, \quad t_1 = m_1 + (m_0 + \overline{r_0}).\overline{r_2}$$

remains to show that $m_0 \vee m_1 \vee m_2$ always true

Compilation: sequential code generation

Single loop (implicit automaton)

```
initializations
forever do
    get inputs
    compute outputs
    update memory
end
```

obvious for data-flow programs:

- sort the variables according to their dependences
  (an order, from the causality criterion in Lustre)
- choose a suitable set of memories

Single loop (1/3)

Example

```
x = 0 -> if edge then
            (pre(x) + y)
         else pre(x) ;
edge = c -> (c and not pre(c)) ;
```

```
_init := true ;
foreach step do
    read(c, y) ;
    if _init then
        { edge := c ; x := 0}
    else
        edge := c and not _pc ;
        if edge then x := x+y ;
    _init := false ; _pc := c ;
end for
```


Single loop (2/3)

Sorting variable computation, and choice of memories

- $x > y$ iff x appears outside of any "pre" in the definition of y (x must be computed after y)
- $>$ is a partial order, if there is no causality error

Single loop (2/3)

Sorting variable computation, and choice of memories

- $x > y$ iff x appears outside of any "pre" in the definition of y (x must be computed after y)
- $>$ is a partial order, if there is no causality error
- $x \succ y$ iff pre(y) appears in the definition of x (x should be computed before y, i.e., from the previous value of y)
- Consider the graph of the relation "$>$"$\cup$"$\succ$".

Single loop (2/3)

Sorting variable computation, and choice of memories

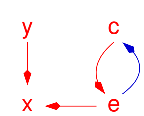
- $x > y$ iff x appears outside of any "pre" in the definition of y (x must be computed after y)
- $>$ is a partial order, if there is no causality error
- $x \succ y$ iff pre(y) appears in the definition of x (x should be computed before y, i.e., from the previous value of y)
- Consider the graph of the relation "$>$"$\cup$"$\succ$".
- Remove $\succ$ edges to cut all loops (if any). The resulting graph is a partial order.
- Whenever an edge $x \succ y$ is removed, a buffer py must be introduced.

Single loop (3/3)

Sorting variable computation, and choice of memories (2)

Example

```
x = 0 -> if edge then
            (pre(x) + y)
         else pre(x) ;
edge = c -> (c and not pre(c)) ;
```




```
_init := true ;
foreach step do
    read(c, y) ;
    if _init then
        { edge := c ; x := 0}
    else
        edge := c and not _pc ;
        if edge then x := x+y ;
    _init := false ; _pc := c ;
end for
```

Explicit control automaton (1/5)

- first way of compiling Esterel
- also applied to data-flow languages
- basis for verification methods (notion of control automaton)

Principle: specialize the code executed at each step (e.g., at initial step)

Explicit control automaton (2/5)

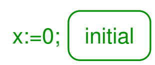
An example in Lustre

```
x = 0 -> if edge then pre(x)+1
         else pre(x);
edge = c ->
      c and not pre(c);
```


Explicit control automaton (2/5)

An example in Lustre

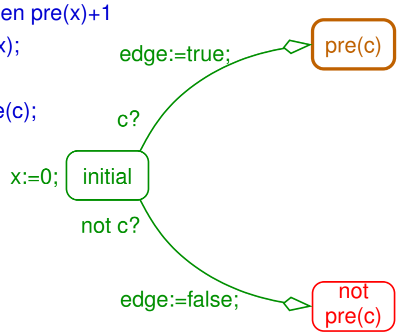
```
x = 0 -> if edge then pre(x)+1
         else pre(x);
edge = c ->
      c and not pre(c);
```



Explicit control automaton (2/5)

An example in Lustre

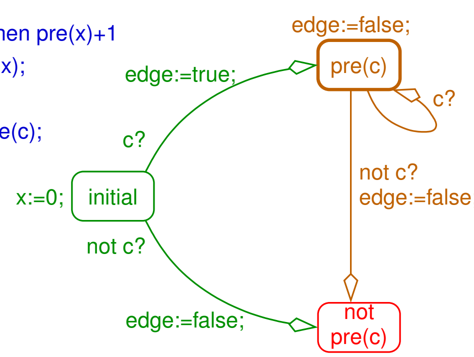
```
x = 0 -> if edge then pre(x)+1
         else pre(x);
edge = c ->
      c and not pre(c);
```



Explicit control automaton (2/5)

An example in Lustre

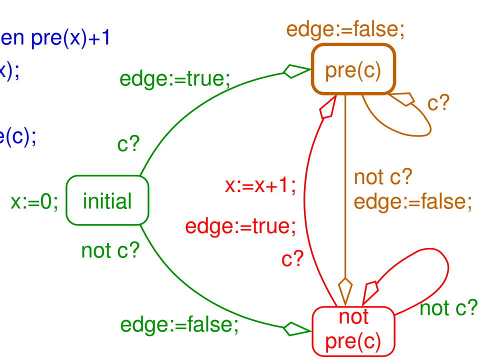
```
x = 0 -> if edge then pre(x)+1
         else pre(x);
edge = c ->
      c and not pre(c);
```



Explicit control automaton (2/5)

An example in Lustre

```
x = 0 -> if edge then pre(x)+1
         else pre(x);
edge = c ->
      c and not pre(c);
```



Explicit control automaton (3/5)

An example in Esterel

```
every R do
   [ await A || await B ] ;
   emit O ;
end every
```

Explicit control automaton (3/5)

An example in Esterel

```
every R do
    [ await A || await B ] ;
    emit O ;
end every
```

In kernel language:

```
abort halt¹ when R;
loop
    abort
        [   abort halt² when A
        ||
            abort halt³ when B
        ];
        emit O; halt⁴
    when R
end
```


Explicit control automaton (4/5)

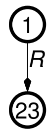
An example in Esterel


```
abort halt¹ when R;
loop
    abort
    [
        abort halt² when A
    ||
        abort halt³ when B
    ];
    emit O; halt⁴
    when R
end
```

DEMO: Boolean mouse

Explicit control automaton (4/5)

An example in Esterel

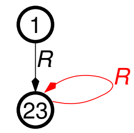



```
abort halt¹ when R;
loop
    abort
    [
        abort halt² when A
    ||
        abort halt³ when B
    ];
    emit O; halt⁴
    when R
end
```

DEMO: Boolean mouse

Explicit control automaton (4/5)

An example in Esterel

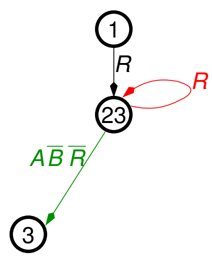



```
abort halt¹ when R;
loop
    abort
    [
        abort halt² when A
    ||
        abort halt³ when B
    ];
    emit O; halt⁴
    when R
end
```

DEMO: Boolean mouse

Explicit control automaton (4/5)

An example in Esterel

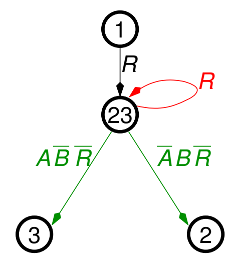



```
abort halt¹ when R;
loop
    abort
    [
        abort halt² when A
    ||
        abort halt³ when B
    ];
    emit O; halt⁴
    when R
end
```

DEMO: Boolean mouse

Explicit control automaton (4/5)

An example in Esterel

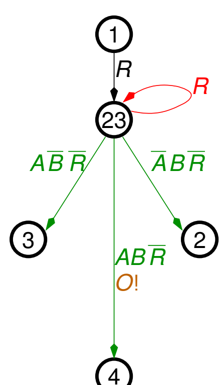



```
abort halt¹ when R;
loop
    abort
    [
        abort halt² when A
    ||
        abort halt³ when B
    ];
    emit O; halt⁴
    when R
end
```

DEMO: Boolean mouse

Explicit control automaton (4/5)

An example in Esterel

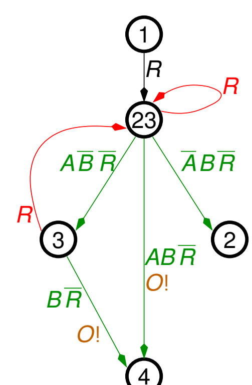



```
abort halt¹ when R;
loop
    abort
    [
        abort halt² when A
    ||
        abort halt³ when B
    ];
    emit O; halt⁴
    when R
end
```

DEMO: Boolean mouse

Explicit control automaton (4/5)

An example in Esterel



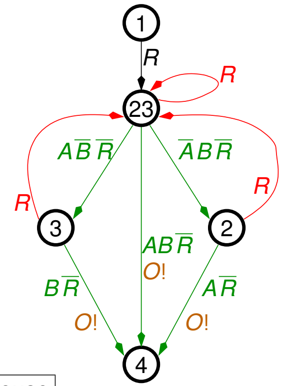

```
abort halt¹ when R;
loop
    abort
    [
        abort halt² when A
    ||
        abort halt³ when B
    ];
    emit O; halt⁴
    when R
end
```

DEMO: Boolean mouse

Explicit control automaton (4/5)

An example in Esterel

```
abort halt1 when R;
loop
    abort
    [
        abort halt2 when A
    ||
        abort halt3 when B
    ];
    emit O; halt4
    when R
end
```

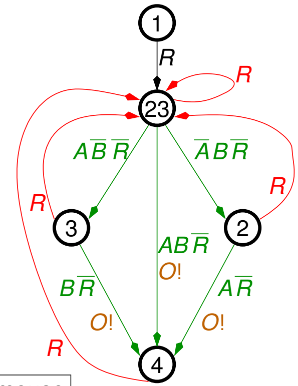


DEMO: Boolean mouse

Explicit control automaton (4/5)

An example in Esterel

```
abort halt1 when R;
loop
    abort
    [
        abort halt2 when A
    ||
        abort halt3 when B
    ];
    emit O; halt4
    when R
end
```



DEMO: Boolean mouse

Explicit control automaton (5/5)

- very efficient code
- possible exponential growth of the code (less than for asynchronous languages)

Esterel is now compiled also into an implicit automaton (single loop)

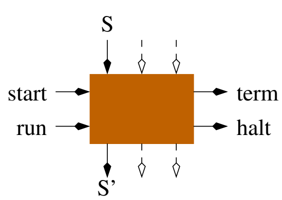
- much more tricky!
- consequence of silicon compiling
- no explosion of the code size

From Esterel to Lustre (1/7)

Principles of the translation of (pure) Esterel into implicit automata (or Esterel → Lustre, or Esterel → circuit)

Each Esterel statement is translated into a node with the same interface:

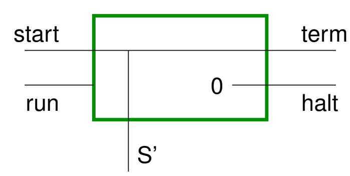
node N
(start, run, S,... : bool)
returns (term, halt, S',...)



From Esterel to Lustre (2/7)

Examples:

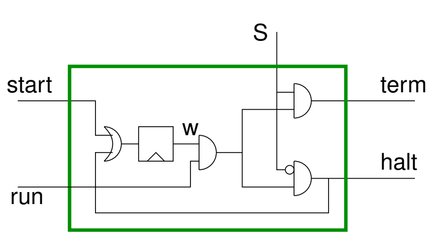
emit S': term = start
halt = false;
S' = start;



From Esterel to Lustre (3/7)

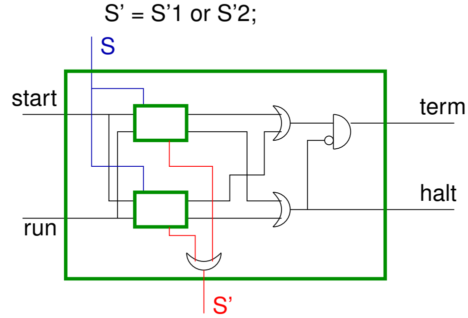
Examples:

await S: term = run and w and S;
halt = run and w and not S;
w = false ->pre(start or halt);



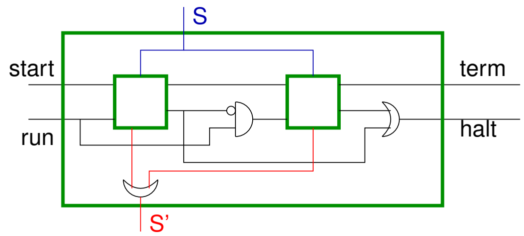
From Esterel to Lustre (4/7)

stat1 || stat2: (term1, halt1, S'1) = Stat1(start,run,...);
(term2, halt2, S'2) = Stat2(start,run,...);
halt = halt1 or halt2;
term = (term1 or term2) and not halt;
S' = S'1 or S'2;



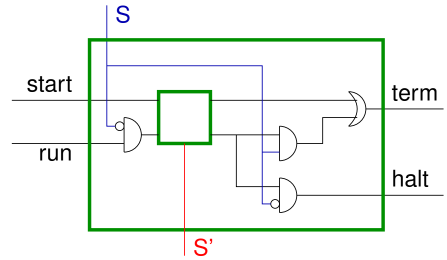
stat1; stat2:

(term1, halt1, S'1) = Stat1(start,run,...);
(term2, halt2, S'2) = Stat2(term1,run and not halt1,...);
halt = halt1 or halt2;
term = term2;
S' = S'1 or S'2;



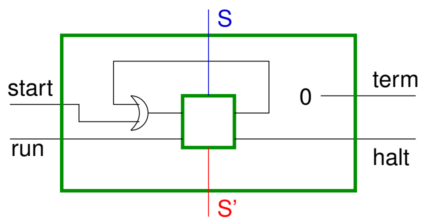
abort stat when S:

(term1, halt1, S',...) = Stat(start,run and not S,...);
halt = halt1 and not S;
term = term1 or (halt1 and S);



loop stat:

(term1, halt1, S',...) = Stat(start or term1, run,...);
term = false;
halt = halt1;



15 Sequential code generation
- Single loop for Lustre
- Explicit Automata
- From Esterel to Lustre
- Reincarnation in Esterel

A last compilation problem: “Reincarnation” in Esterel (1/4)

```
loop
   signal S in
      [
         await T ; emit S
      ||
         present S then emit O
      ]
   end
end
```


A last compilation problem: “Reincarnation” in Esterel (1/4)

```
loop
   signal S in
      [
         await T ; emit S
      ||
         present S then emit O
      ]
   end
end
```

tick

A last compilation problem: “Reincarnation” in Esterel (1/4)

```
loop
   signal S in
      [
         await T ; emit S
      ||
         present S then emit O
      ]
   end
end
```

tick

T S

A last compilation problem: “Reincarnation” in Esterel (1/4)

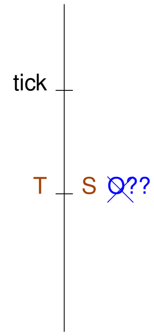
```
loop
   signal S in
      [
         await T ; emit S
      ||
         present S then emit O
      ]
   end
end
```

tick

T S O??

A last compilation problem: "Reincarnation" in Esterel (1/4)

```
loop
   signal S in
      [
         await T ; emit S
      ||
         present S then emit O
      ]
   end
end
```



"Reincarnation" in Esterel (2/4)

Solution1 : Code replication

```
loop
   signal S1 in   [   await T; emit S1
                  ||  present S1 then emit O
                  ]
   end
   signal S2 in   [   await T; emit S2
                  ||  present S2 then emit O
                  ]
   end
end
```


"Reincarnation" in Esterel (3/4)

Solution 2 : Distinguish between

- the "surface" of the body, i.e., a piece of code corresponding to its first reaction,
- and its "depth", i.e., a piece of code corresponding to other reactions.

"Reincarnation" in Esterel (3/4)

Solution 2 : Distinguish between

- the "surface" of the body, i.e., a piece of code corresponding to its first reaction,
- and its "depth", i.e., a piece of code corresponding to other reactions.

surf(await T) = pause
depth(await T) = await immediate T
surf(present S then emit O) = present S then emit O
depth(present S then emit O) = nothing

"Reincarnation" in Esterel (4/4)

```
loop
   signal S1 in   [   pause
                  ||  present S1 then emit O
                  ]
   end
   signal S2 in   [   await immediate T; emit S2
                  ||  nothing
                  ]
   end
end
```


Verification and Automatic Testing of Synchronous Programs

Validation of Reactive Systems (1/2)

- crucial goal
- critical properties (safety)
  "something (bad) never happens"
- the environment must be taken into account

Validation of Reactive Systems (2/2)

Two complementary techniques

- Formal verification
- Testing

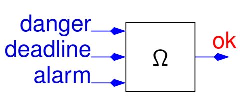
Both require the formal description of

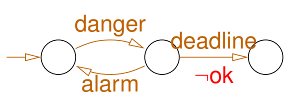
- the expected behavior of the system (properties)
- the assumed behavior of the environment (assumptions)

Expressing Properties using Synchronous Observers

A program observing relevant variables, and computing a Boolean output, true as long as the property is satisfied (safety).

Example – "any occurrence of "danger" is followed by an "alarm" before the next "deadline".





Synchronous Observers (1/3)

An observer in Lustre

```
node Omega (danger, alarm, deadline: bool)
    returns (ok: bool);
var alarm_since_danger: bool
let
    ok = deadline => alarm_since_danger;
    alarm_since_danger =
        if alarm then true
        else if danger then false
        else (true -> pre(alarm_since_danger));
tel
```

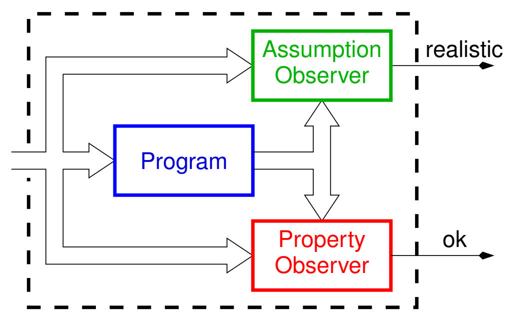

Synchronous Observers (2/3)

Advantages

- Transform path properties into state properties
- Same language to write programs and properties
- Observers are executable
  - they can be tested
  - they can be executed on-line, concurrently with the program, during its actual execution (self-test, redundancy, monitoring, "runtime verification")

Synchronous Observers (3/3)

"Verification Program"



Check: "if ok=false then realistic=false before"

Formal Verification

Synchronous programs as Interpreted Automata

- Split the program into a finite control part and a data part.
- In Lustre, the control part may be obtained by a restriction to Boolean (or finite state) variables (partial evaluation).
- Result: an explicit or implicit interpreted automaton

Aleady used in compilation (Esterel and Lustre)

18 Verification
- Model-checking
- Numerical properties

Verification by Model-Checking

Exhaustive exploration of the control part of the automaton (Boolean abstraction): build all the states that can be reached from the initial state with relevant always true, checking that ok is always true.

- exact for purely Boolean programs
- conservative for general programs

Model-Checking: Example

The Switch example in Lustre:

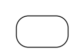
```
node TwoStates1
    (init, set, reset: bool)
returns (state: bool);
let
    state = init ->
        if set then true
        else if reset then false
        else pre(state);
tel
```

```
node TwoStates2
    (init, set, reset: bool)
returns (state: bool);
let
    state = init ->
        if set and not pre(state)
            then true
        else if reset and pre(state)
            then false
        else pre(state);
tel
```

Show that they behave the same if set and reset are never true together

Model-Checking: Example

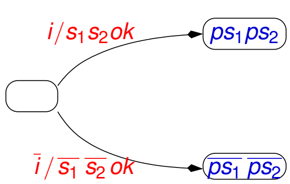
```
node Verif(init, set, reset: bool) returns (ok: bool);
var s1, s2: bool;
let s1 = TwoStates1(init, set, reset);
    s2 = TwoStates2(init, set, reset);
    ok = (s1=s2);
tel
```



DEMO

Model-Checking: Example

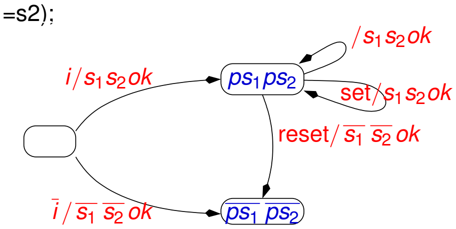
```
node Verif(init, set, reset: bool) returns (ok: bool);
var s1, s2: bool;
let s1 = TwoStates1(init, set, reset);
    s2 = TwoStates2(init, set, reset);
    ok = (s1=s2);
tel
```



DEMO

Model-Checking: Example

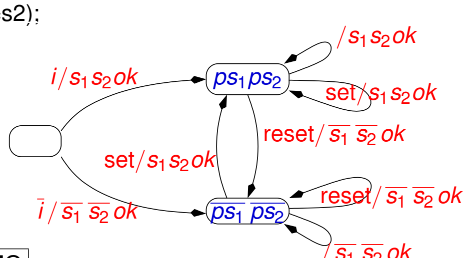
```
node Verif(init, set, reset: bool) returns (ok: bool);
var s1, s2: bool;
let s1 = TwoStates1(init, set, reset);
    s2 = TwoStates2(init, set, reset);
    ok = (s1=s2);
tel
```



DEMO

Model-Checking: Example

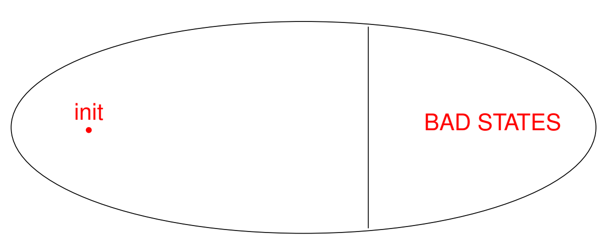
```
node Verif(init, set, reset: bool) returns (ok: bool);
var s1, s2: bool;
let s1 = TwoStates1(init, set, reset);
    s2 = TwoStates2(init, set, reset);
    ok = (s1=s2);
tel
```



DEMO

LESAR: a verification tool.

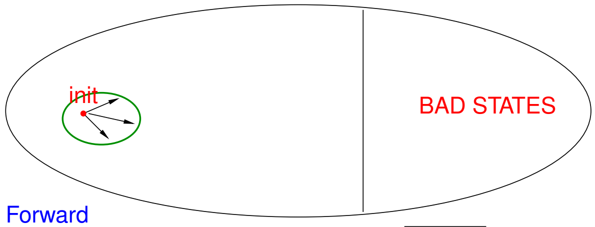
- several engines: enumerative search, symbolic (BDD-based) forward or backward search



DEMO

LESAR: a verification tool.

- several engines: enumerative search, symbolic (BDD-based) forward or backward search

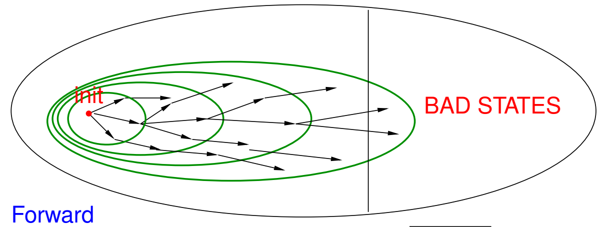


Forward

DEMO

LESAR: a verification tool.

- several engines: enumerative search, symbolic (BDD-based) forward or backward search

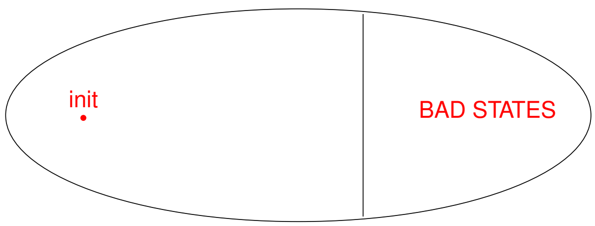


Forward

DEMO

LESAR: a verification tool.

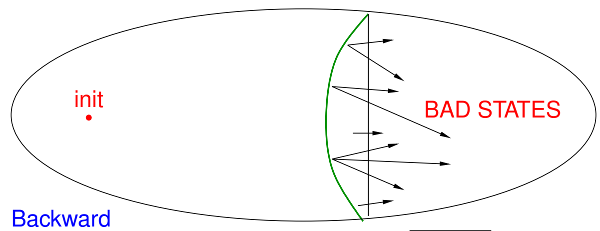
- several engines: enumerative search, symbolic (BDD-based) forward or backward search



DEMO

LESAR: a verification tool.

- several engines: enumerative search, symbolic (BDD-based) forward or backward search

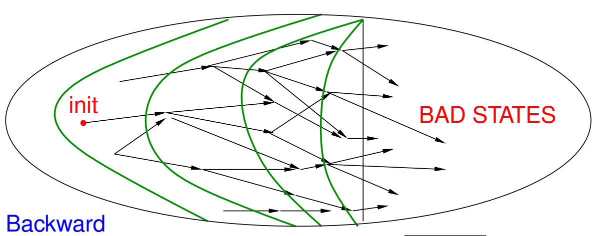


Backward

DEMO

LESAR: a verification tool.

- several engines: enumerative search, symbolic (BDD-based) forward or backward search



Backward

DEMO

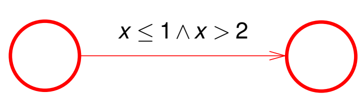
Numerical Properties

Approaches based on the control automaton only provide exact results when applied to pure control programs (Pure Esterel, Boolean Lustre, . . .)
Interpreted Automata (with data): approximate, conservative results

Some unfeasible transitions are considered

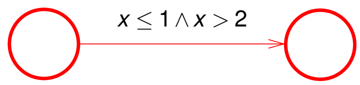
Unfeasible transitions

Statically unfeasible transitions

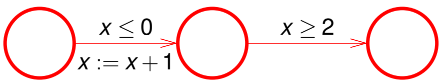


Unfeasible transitions

Statically unfeasible transitions



Dynamically unfeasible transitions



Example: A subway speed regulation system

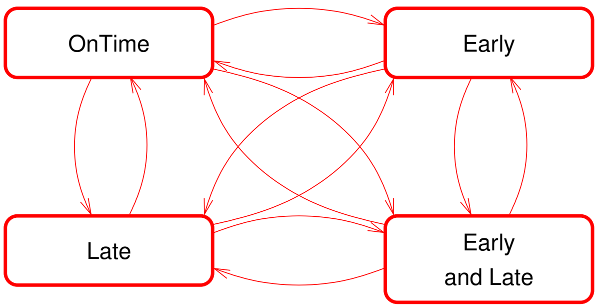
Each train counts
- the number #b of beacons encountered along the track
- the number #s of seconds received from a central clock

Ideally, a train should meet a beacon each second.

In fact,
- When #b becomes ≥#s+10, the train is considered early, until #b≤#s (hysteresis)
- When #s becomes ≥#b+10, the train is considered late, until #s≤#b

A subway speed regulation system (2/5)

Interpreted automaton



A subway speed regulation system (3/5)

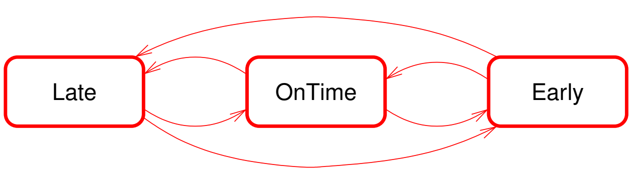
Code of state: "On_time"

```
if BEACON then DIFF:=DIFF+1
else if SECOND then DIFF:=DIFF−1;
if DIFF<−10 then
    if DIFF>10 then goto "Early_and_Late"
    else goto "Late"
else if DIFF>10 then goto "Early"
else goto "On_time"
```


A subway speed regulation system (4/5)

Removing statically unfeasible transitions
- Use assumptions (boring and error-prone)
- Satisfiability of linear constraints can be easily checked

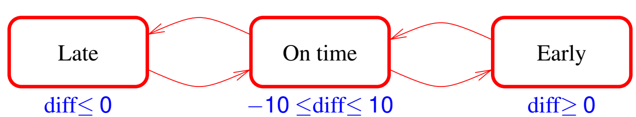
Result on the example



A subway speed regulation system (5/5)

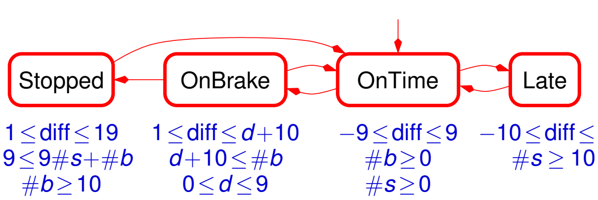
Removing dynamically unfeasible transitions
- Use abstract interpretation to build, in each state of the automaton, an upper approximation (system of linear constraints) of the set of variable valuations when the control is at this state.

Result on the example



The complete subway example

- When a train is early, it puts on brakes; continuously braking makes the train stop before encountering 10 beacons.
- When a train is late, it warns the central clock, which stops sending the SECOND signal until no train is late.



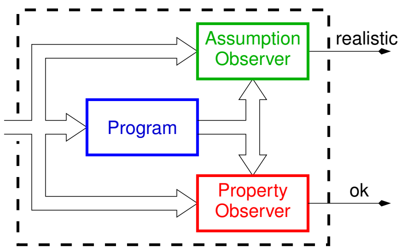
The complete subway example (2 trains)

For two trains, we get $|\#b_1 - \#b_2| \leq 29$, showing that two trains initially separated by more that 29 beacons will never collide.

Testing

- Strong industrial demand
- To be used when verification fails
  - Too complex programs (numbers)
  - Black-box programs
    (partly described in low-level languages, distributed implementations)

Automatic Testing



- use the assumption observer to generate realistic test sequences
- use the property observer as an "oracle" to analyze the results of the test

Automatic Testing: Example (1/3)

- The controller C should maintain the temperature u between 17° and 20° by turning a heater on and off.
- When the heater is running, the temperature u increases with $0.2 \leq du/dt \leq 0.5$; otherwise it decreases with $-0.3 \leq du/dt \leq -0.1$.
- Initially, the heater is turned off, and $18^\circ \leq u \leq 19^\circ$

Automatic Testing: Example (2/3)

```
node Assumption(on, off: bool; u: real) returns (realistic: bool);
var dudt: real; heating: bool;
let
   realistic = (u>=18 and u<=19) ->
               if heating then (dudt>=0.2 and dudt<=0.5)
               else (dudt>=–0.3 and dudt<=–0.1);
   dudt = (u – pre(u));
   heating = false -> if pre(on) then true
                      else if pre(off) then false
                      else pre(heating);
tel
```


Automatic Testing: Example (3/3)

```
node Property(on, off: bool; u: real) returns (ok: bool);
let
   ok = (u>=17 and u<=20);
tel
```


Testing tool: Lurette

Given

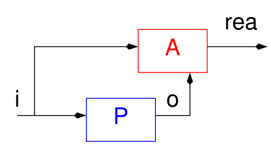
- the observers "Assumption" and "Property"
- the executable code of the program under test

generate and run arbitrarily long test sequences, satisfying the "Assumption", while checking the "Property"

Generation of input sequences (1/3)

Realistic test sequences

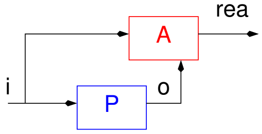
$(i_1, i_2, \ldots, i_n)$ involving outputs
$(\langle o_1, \mathrm{rea}_1\rangle, \langle o_2, \mathrm{rea}_2\rangle, \ldots, \langle o_n, \mathrm{rea}_n\rangle)$ with $\mathrm{rea}_i = \mathit{true}, i = 1..n$.



Generation of input sequences (2/3)

First attempt: Observer as an acceptor

- randomly choose input i
- get the corresp. output o
- submit (i, o) to the observer
- in case of refusal choose again.



Two problems:

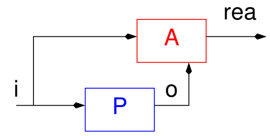
- the observer needs the program output o
- the probability to randomly choose a realistic input can be very small (e.g., $X = Y$)

Generation of input sequences (3/3)

Second idea: Observer as an generator

Global state: $S = \langle S_P, S_A \rangle$

$\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)$



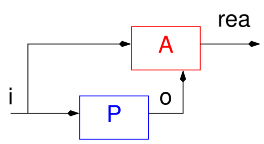
$\langle S_P, S_A \rangle$

Generation of input sequences (3/3)

Second idea: Observer as an generator

Global state: $S = \langle S_P, S_A \rangle$

$\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)$



$\langle S_P, S_A \rangle$

choice of i

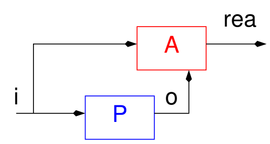
(s.t. $f(S_A, i) =$true)

Generation of input sequences (3/3)

Second idea: Observer as an generator

Global state: $S = \langle S_P, S_A \rangle$

$\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)$



$\langle S_P, S_A \rangle$

choice of i

(s.t. $f(S_A, i) =$true)

$S_P \xrightarrow{i/o} S'_P$

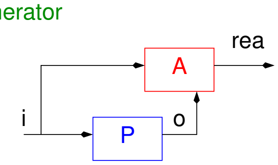
running P

Generation of input sequences (3/3)

Second idea: Observer as an generator

Global state: $S = \langle S_P, S_A \rangle$

$\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)$



$\langle S_P, S_A \rangle$

choice of i

(s.t. $f(S_A, i) =$true)

$S_P \xrightarrow{i/o} S'_P$

running P

$S_A \xrightarrow{(i,o)/\text{true}} S'_A$

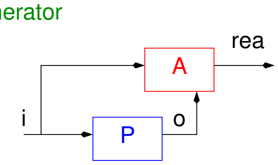
running A

Generation of input sequences (3/3)

Second idea: Observer as an generator

Global state: $S = \langle S_P, S_A \rangle$

$\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)$



$\langle S_P, S_A \rangle$ $\langle S'_P, S'_A \rangle$

choice of i

(s.t. $f(S_A, i) =$true)

$S_P \xrightarrow{i/o} S'_P$

running P

$S_A \xrightarrow{(i,o)/\text{true}} S'_A$

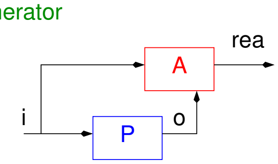
running A

Generation of input sequences (3/3)

Second idea: Observer as an generator

Global state: $S = \langle S_P, S_A \rangle$

$\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)$



$\langle S_P, S_A \rangle \longrightarrow \langle S'_P, S'_A \rangle$

choice of i

(s.t. $f(S_A, i) =$true)

$S_P \xrightarrow{i/o} S'_P$

running P

$S_A \xrightarrow{(i,o)/\text{true}} S'_A$

running A

Example (1/2)

First step

```
realistic = (u>=18 and u<=19) ->
                    if heating then (dudt>=0.2 and dudt<=0.5)
                    else (dudt>=–0.3 and dudt<=–0.1);
dudt = (u – pre(u));
heating = false -> if pre(on) then true
                   else if pre(off) then false
                   else pre(heating);
```

Example (1/2)

First step

realistic = (u>=18 and u<=19) ->
if heating then (dudt>=0.2 and dudt<=0.5)
else (dudt>=–0.3 and dudt<=–0.1);
dudt = (u – pre(u));
heating = false -> if pre(on) then true
else if pre(off) then false
else pre(heating);

Select, e.g., u=18, run a program step, get the output, e.g., on=off=false

Example (2/2)

Second step:
You know pre(u)=18, pre(on)=pre(off)=pre(heating)=false

Automatic Testing

Example (2/2)

Second step:
You know pre(u)=18, pre(on)=pre(off)=pre(heating)=false

dudt = (u – pre(u)); i.e., u–18
heating = false -> if pre(on) then true
else if pre(off) then false
else pre(heating); i.e., false

N. Halbwachs (Verimag/CNRS) Synchronous Programming 197 / 198

Automatic Testing

Example (2/2)

Second step:
You know pre(u)=18, pre(on)=pre(off)=pre(heating)=false

realistic = (u>=18 and u<=19) ->
if heating then (dudt>=0.2 and dudt<=0.5)
else (dudt>=–0.3 and dudt<=–0.1);
dudt = (u – pre(u)); i.e., u–18
heating = false -> if pre(on) then true
else if pre(off) then false
else pre(heating); i.e., false

You get $-0.3 \leq u-18 \leq -0.1$, i.e., $17.7 \leq u \leq 17.9$.
Select, e.g., u=17.7, run a program step, and so on. . .

N. Halbwachs (Verimag/CNRS) Synchronous Programming 197 / 198

Conclusion

Both for formal verification and for automatic testing, the user has to provide the same information:

- an observer specifying properties of interest (safety)
- an observer specifying known assumptions about the environment

Automatic tools are then available to make the job

Presently, we also have a specific language to express input scenarios (non deterministic, constrained)

A short bibliography

1 General principles, basic papers and books

References

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[BCG88] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems, an introduction to ESTEREL. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*. Elsevier Science Publisher B.V. (North Holland), 1988. INRIA Report 647.

[BCGH93] A. Benveniste, P. Caspi, P. Le Guernic, and N. Halbwachs. Data-flow synchronous languages. In REX *Symposium "Ten Years of Concurrency, Reflections and Perspectives"*. LNCS 803, Springer-Verlag, June 1993.

[Ber89] G. Berry. Real time programming: Special purpose or general purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.

[Ber93] G. Berry. Preemption and concurrency. In *Proc. FSTTCS 93*, Lecture Notes in Computer Science 761, pages 72–93. Springer-Verlag, 1993.

[Cas92] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[Hal98] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[PMM+98] A. Poigné, M. Morley, O. Maffeis, L. Hoelenderski, and R. Budde. The synchronous approach to design reactive systems. *Formal Methods in System Design*, 12:163–187, 1998.

2 Synchronous languages

References

[And96] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, July 1996.

[BC84] G. Berry and L. Cosserat. The synchronous programming language Esterel and its mathematical semantics. In S. Brookes LNCS 197 and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer Verlag, 1984.

[Ber95] G. Berry. The constructive semantics of Esterel. Draft book available by ftp at `ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness.ps.gz`, 1995.

[BG90] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BS91] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, POPL'87*, Munchen, January 1987.

[GBBG86] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL, a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362–374, 1986.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[LGLL91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, Stony Brook, August 1992. LNCS 630, Springer Verlag.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer Verlag.

[RR02] P. Raymond and Y. Roux. Describing non-deterministic reactive systems by means of regular expressions. In *First Workshop on Synchronous Languages, Applications and Programming, SLAP'02*, Grenoble, April 2002.

3 Compilation

References

[Ber91] G. Berry. A hardware implementation of pure ESTEREL. In *ACM Workshop on Formal Methods in VLSI Design, Miami*, January 1991.

[Edw02] S. A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits ans Systems*, 21, 2002.

[HM95] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, Como (Italy), September 1995.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991. LNCS 528, Springer Verlag.

[MH96] F. Maraninchi and N. Halbwachs. Compiling Argos into boolean equations. In *Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT)*, Uppsala (Sweden), September 1996. LNCS 1135, Springer Verlag.

[SBT96] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *International Design and Testing Conference IDTC'96*, Paris, France, 1996.

[WBC+00] D. Weil, V. Bertin, E. Closse, M. Poisse, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, 2000.

4 Verification and Validation

References

[Bou98] A. Bouali. Xeve: an Esterel verification environment. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[DR94] R. De Simone and A. Ressouche. Compositional semantics of ESTEREL and verification by compositional reductions. In D. Dill, editor, *6th International Conference on Computer Aided Verification, CAV'94*, Stanford, June 1994. LNCS 818, Springer Verlag.

[dSR94] R. de Simone and Annie Ressouche. Compositional semantics of ESTEREL and verification by compositional reductions. In *CAV'94*, Stanford, June 1994.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, pages 785–793, September 1992.

[HPOG89] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.C. Glory. Specifying, programming and verifying real-time systems, using a synchronous declarative language. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, June 1989.

[JJGM03] E. Jahier, B. Jeannet, F. Gaucher, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *AADEBUG'2003 – Fifth International Workshop on Automated Debugging*, Ghent, September 2003.

[JRB04] E. Jahier, P. Raymond, and P. Baufreton. Case studies with Lurette V2. In *First International Symposium on Leveraging Applications of Formal Method, ISoLa 2004*, Paphos, Cyprus, October 2004.

[LDBL93] M. Le Borgne, Bruno Dutertre, Albert Benveniste, and Paul Le Guernic. Dynamical systems over Galois fields. In *European Control Conference*, pages 2191–2196, Groningen, 1993.

[MM04] F. Maraninchi and L. Morel. Arrays and contracts for the specification and analysis of regular systems. In *Fourth International Conference on Application of Concurrency to System Design (ACSD)*, Hamilton, Ontario, Canada, June 2004.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

5 Synchrony/Asynchrony, Code distribution

References

[BCG99] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99*. LNCS 1664, Springer Verlag, 1999.

[BCT99] A. Benveniste, P. Caspi, and S. Tripakis. Distributing synchronous programs on a loosely synchronous, distributed architecture. Research Report 1289, Irisa, December 1999.

[BS98] G. Berry and E. Sentovich. Embedding synchronous circuits in GALS-based systems. In *Sophia-Antipolis conference on Micro-Electronics (SAME 98)*, October 1998.

[BS01] G. Berry and E. Sentovich. Multiclock Esterel. In *Correct Hardware Design and Verification Methods, CHARME'01*, Livingston (Scotland), September 2001. LNCS 2144, Springer Verlag.

[BSR93] G. Berry, R. K. Shyamasundar, and S. Ramesh. Communicating reactive processes. In *Proc. 20th ACM Conf. on Principles of Programming Languages, POPL'93*, Charleston, Virginia, 1993.

[Cas01] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *1st International Workshop on Embedded Software, EMSOFT2001*, Lake Tahoe, October 2001. LNCS 2211.

[CCM+03] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to Scade/Lustre to TTA: A layered approach for distributed embedded applications. In *LCTES 2003*, San Diego, CA, June 2003.

[CGP94] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, USA, October 1994. ISCA.

[CMR01] P. Caspi, C. Mazuet, and N. Reynaud Paligot. About the design of distributed control systems, the quasi-synchronous approach. In *SAFECOMP'01*. LNCS 2187, 2001.

[CS00] P. Caspi and R. Salem. Threshold and bounded-delay voting in critical control systems. In *FTRTFT'2000*, Pune, India, September 2000. LNCS 1926.

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*. LNCS 2491, Springer Verlag, October 2002.

[ML94] O. Maffeis and P. Le Guernic. Distributed implementation of SIGNAL: scheduling and graph clustering. In *3rd Internationl Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. LNCS 863, Springer Verlag, September 1994.

[SC04] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, June 2004.

6 Reactive programming, Higher order languages

References

[BdS96] F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Transactions on Software Engineering*, 22(4):256–266, April 1996.

[Bou91] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[BS98] F. Boussinot and J.-F. Susini. The SugarCubes tool box - a reactive Java framework. *Software Practice and Experience*, 28(14):1531–1550, 1998.

[BS00] F. Boussinot and J.-F. Susini. Java threads and SugarCubes. *Software Practice and Experience*, 30(14):545–566, 2000.

[Cas93] P. Caspi. Lucid Synchrone. In *International Workshop on Principles of Parallel Computing (OPOPAC)*, November 1993.

[CDE+06] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-Synchronous Kahn Networks: a relaxed model of synchrony for real-time systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston (N.C.), January 2006.

[CP95] P. Caspi and M. Pouzet. A functional extension to LUSTRE. In *Eighth International Symp. on Languages for Intensional Programming, ISLIP'95*, Sidney, May 1995.

[CP96] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Int. Conf. on Functional Programming, Philadelphia*. ACM SIGPLAN, May 1996.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[MP05] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*, Lisboa, July 2005.