Declarative Synchronous Languages

Lustre / Signal

Languages

Say what IS or what SHOULD BE

Declarative languages

Imperative langages

Say what MUST BE DONE

LUSTRE/SIGNAL

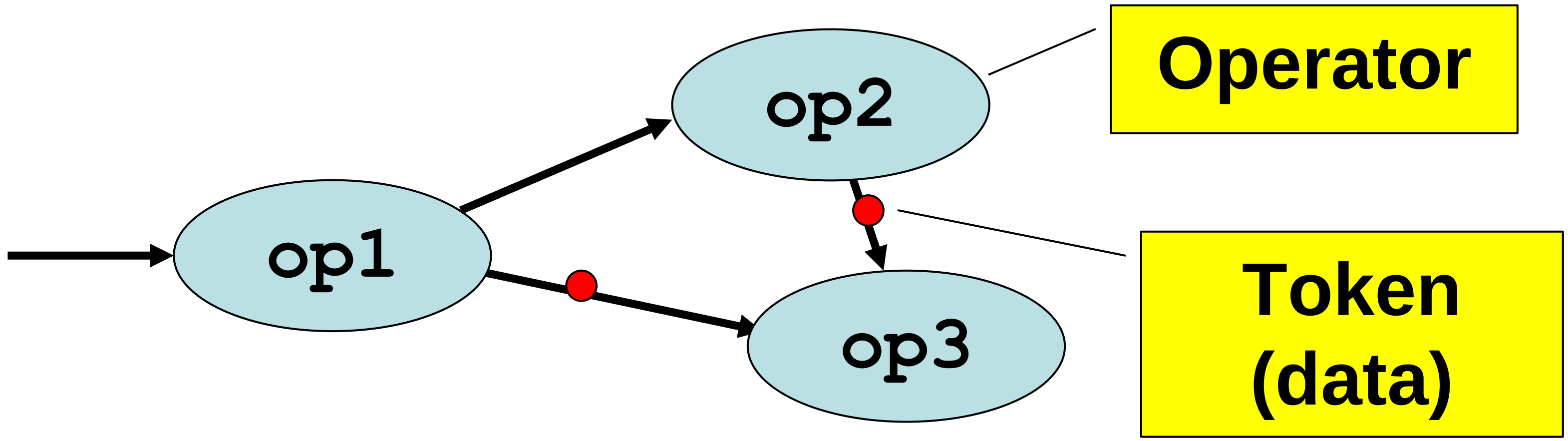
Language	Characteristics	Applications
LUSTRE (and the associated tool SCADE)	Functional Mono clock	Critical systems; Command/Control; Circuits
SIGNAL	Relational (constraints) Multiple-clocked	System specification. Automatic control; Signal Data Processing

LUSTRE

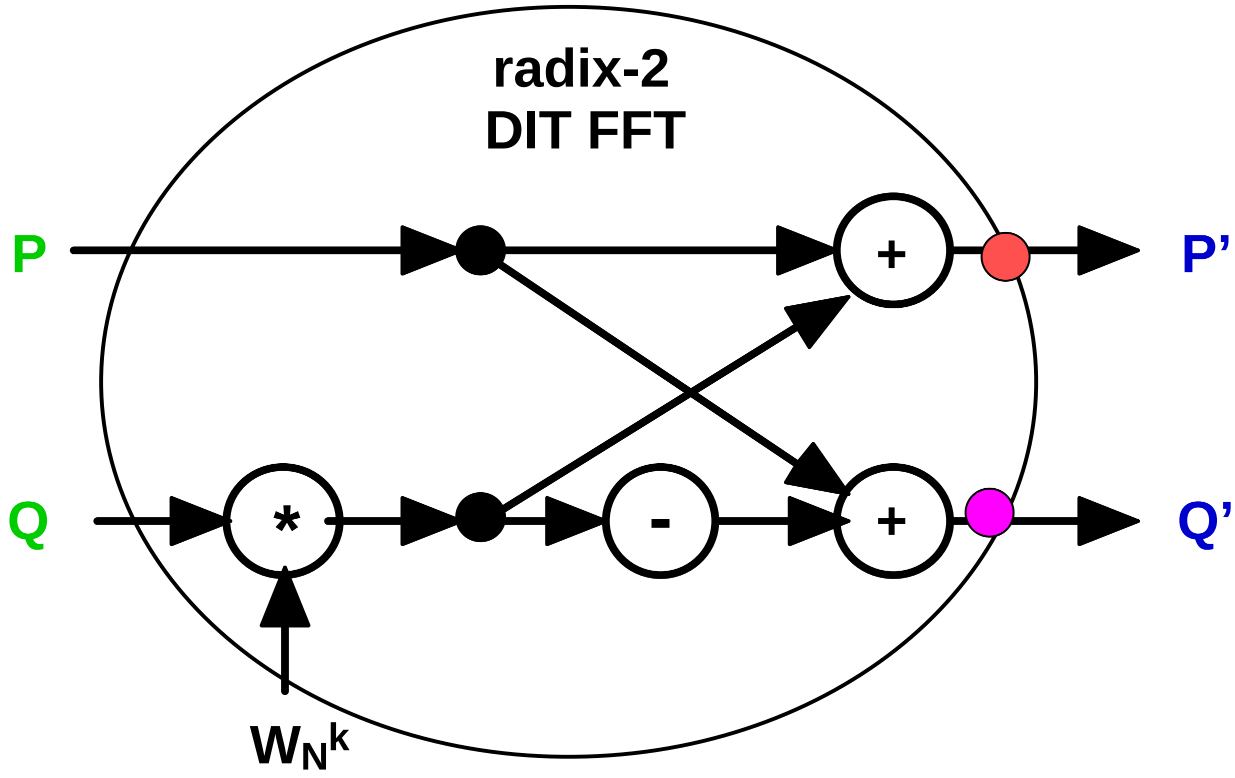
- We study LUSTRE in depth because
  - It is a very simple language (4 primitive operators to express reactions)
  - Relies on models familiar to engineers
    - Equation systems
    - Data flow network
  - Lends itself to formal verification (it is a kind of logical language)
  - Very simple (mathematical) semantics

Operator Networks

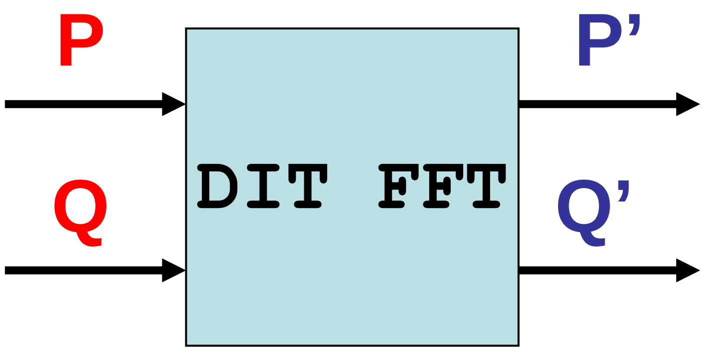
- LUSTRE and SIGNAL programs can be interpreted as networks of operators.
- Data « flow » to operators where they are consumed. Then, the operators generate new data. (Data Flow description).



Data Flow



Functional Point of View



$$P' = P + W_N^k * Q$$

$$Q' = P - W_N^k * Q$$

LUSTRE

Flows, Clocks

- A flow is a pair made of
  - A possibly infinite sequence of values of a given type
  - A clock representing a sequence of instants

$$\mathbf{X{:}T} \qquad (\mathbf{x_1, x_2, \ldots, x_n, \ldots})$$

Language (1)

Variable :

- typed
- If not an input variable, defined by 1 and only 1 equation
- Predefined types: **`int`**, **`bool`**, **`real`**
- tuples: **`(a,b,c)`**

Equation :

`X = E` means $\forall \mathbf{k}, \mathbf{x}_k = \mathbf{e}_k$

Assertion :

Boolean expression that should be always **`true`** at each instant of its clock.

Language (2)

Substitution principle:

if `X = E` then `E` can be substituted for `X` anywhere in the program and conversely

Definition principle:

A variable is fully defined by its declaration and the equation in which it appears as a left-hand side term

Expressions

Constants

`0, 1, ..., true, false, ..., 1.52, ...`

`int` `bool` `real`

+ Imported types and operators

$$c : \alpha \Leftrightarrow \forall k \in \mathbb{N}, c_k = c$$

Operators and Types

- Strict operators:

  $\texttt{mod} : \texttt{int} \times \texttt{int} \to \texttt{int}$

- Loaded operators:

  $+ : \texttt{int} \times \texttt{int} \to \texttt{int}$

  $+ : \texttt{real} \times \texttt{real} \to \texttt{real}$

- Polymorphic operators:

  $\texttt{<>} : \alpha \times \alpha \to \texttt{bool}$

« Combinational » Lustre

Data operators

Arithmetical: **+**, **–**, *****, **/**, **div**, **mod**

Logical: **and**, **or**, **not**, **xor**, **=>**

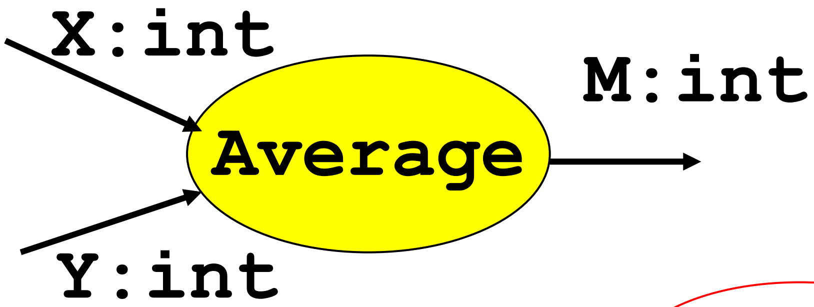
Conditional: **if** … **then** … **else** …

Casts: **int**, **real**

« Point-wise » operators

$$X \; op \; Y \Leftrightarrow \forall k, (X \; op \; Y)_k = X_k \; op \; Y_k$$

« Combinational » Example



Operator

Data

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

Result

Definition

$$\forall k \in \mathbb{N}, M_k = (X_k + Y_k) / 2_k$$

Example (suite)

```
node Average (X,Y:int)
  returns (M:int);
var S:int;   -- local variable
let
  S = X + Y; -- non significant order
  M = S / 2;
tel
```

By substitution, the behavior is the same

Memorizing

Take the past into account!

`pre` (previous):

$$X = (x_1, x_2, \cdots, x_n, \cdots) \ : \ pre(X) = \left(\text{nil}, x_1, \cdots, x_{n-1}, \cdots\right)$$

Undefined value denoting uninitialized memory: nil

`->` (initialize): sometimes call “followed by”

$$X = (x_1, x_2, \cdots, x_n, \ldots) \ , \ Y = (y_1, y_2, \cdots, y_n, \ldots) \ :$$

$$(X -> Y) = (x_1, y_2, \cdots, y_n, \ldots)$$

« Sequential » Examples

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

$$E_1 = false; \forall k > 1, E_k = \overline{X_{k-1}} \wedge X_k$$

Exercice: Modify to capture possible initial edge

tuple

```
node MinMax (X:int) returns (min,max:int);
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

Temporal recursion

Usual. Use **pre** and **->**

e.g.: `nat = 1 -> pre nat + 1`

Instantaneous recursion

e.g.: `X = 1.0 / (2.0 – X)`

Forbidden in Lustre, even if a solution exists!

Be carefull with cross-recursion.

Clocks

Basic clock

Discrete time induced by the input sequence

Derived clocks (slower)

`when` (filter operator):

`E when C` is the sub-sequence of **E** obtained by keeping only the values of indexes e_k for which c_k**=true**

Examples of clocks

Basic cycles	1	2	3	4	5	6	7	8
C1	true	false	true	true	false	true	false	true
Cycles of C1	1		2	3		4		5
C2	false		true	false		true		true
Cycles of C2			1			2		3

Example of sampling

For

```
nat,odd:int
halfBaseClock:bool
nat = 1 -> pre nat +1;
halfBaseClock =
  true -> not pre halfBaseClock;
odd = nat when halfBaseClock;
```

`nat` is a flow on the basic clock;
`odd` is a flow on `halfBaseClock`

Exercice: write `even`

Interpolation operator

« converse » of sampling

`current` (interpolation) :

Let **`E`** be an expression whose clock is `C`, **`current(E)`** is an expression on the clock of `C`, and its value at any instant of this clock is the value of **`E`** at the last time when **`c`** was **`true`**.

`current (X when C)` $\neq$ **`X`**

`current` can yield **`nil`**

Example of current

Basic cycles	1	2	3	4	5	6	7
C	ff	tt	ff	tt	ff	ff	tt
X	x1	x2	x3	x4	x5	x6	x7
Y = X when C		x2		x4			x7
Z = current(Y)	nil	x2	x2	x4	x4	x4	x7

Other examples of current

X	1	2	3	4	5	6	7
Y	t	f	t	t	t	f	f
C	t	t	f	t	t	f	t
Z=X when C	1	2		4	5		7
H=Y when C	t	f		t	t		f
T=Z when H	1			4	5		
current T	1	1		4	5		5
current (current T)	1	1	1	4	5	5	5

The initialization issue

`Y=current(X when C)` where C_1 = false is erroneous.

Possible solutions:

- Strict discipline: ensure that C_1 is always true.
- Force the clock to true at the first instant:

```
CC = true -> C; Y=current(X when CC);
```

- Provide a default value D :

```
Y = if C then current(X when C) else
         D -> pre Y;
```

First programs

Counters

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

C	tt	ff	tt	tt	ff	ff	tt
COUNTER(0,2,false)	0	2	4	6	8	10	12
COUNTER((0,2,false)when C)	0		2	4			6
COUNTER(0,2,false)when C	0		4	6			12

Edges

```
node Edge (b:bool) returns (f:bool);
-- detection of a rising edge
let
    f = false -> (b and not pre(b));
tel;
```

initial

Undefined at the first instant

```
Falling_Edge = Edge(not c);
```

Retriggerable Monostable

```
node MR(set:bool;delay:int) returns (z:bool);
-- retriggerable monostable
var n: int;
let
    z = ( n>0 );
    n = 0 -> if set then
                 delay
             else
                 if pre(n) = 0 then
                     0
                 else
                     pre(n) - 1;
tel;
```

Local variable

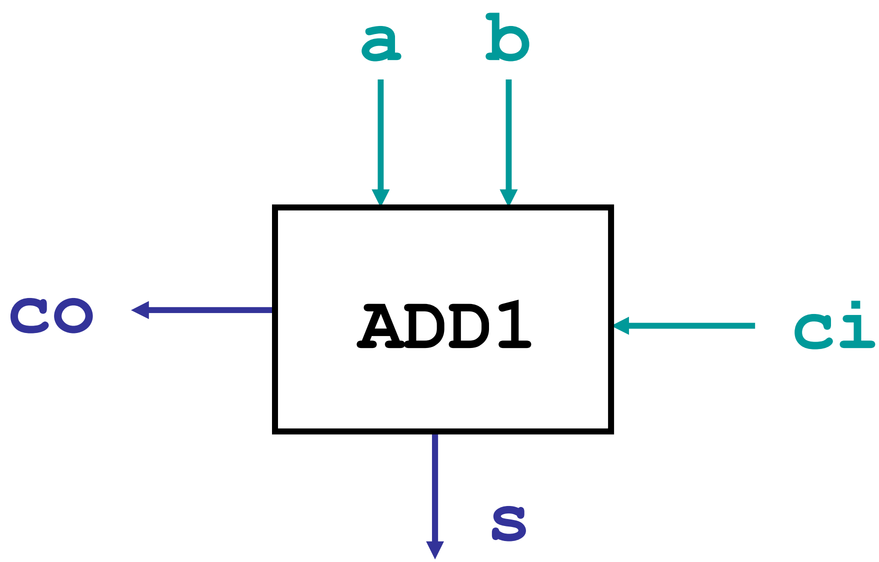
2 equations: **non significant order**

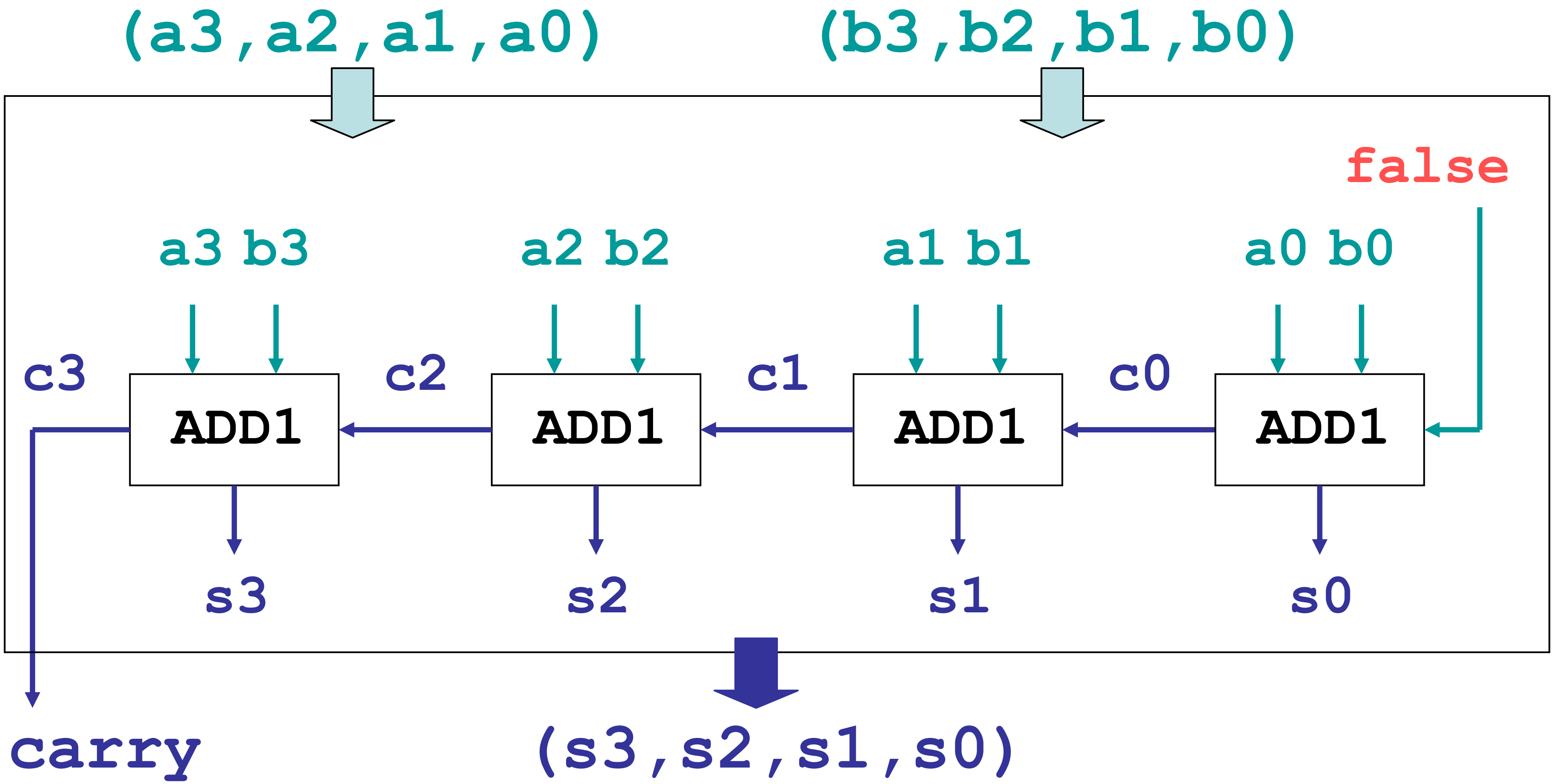
Non Retriggerable Monostable

```
node MNR ( e:bool; p:int )
                         returns (s:bool);
var set:bool;
let
    s = MR (set,p);
    set = Edge(e) and not pre(s);
tel;
```

Adder

```
node ADD1 (a,b,ci:bool)
          returns (s,co:bool);
let
    s = ci xor a xor b;
    co = (b and ci) or (ci and a)
           or (a and b);
tel
```

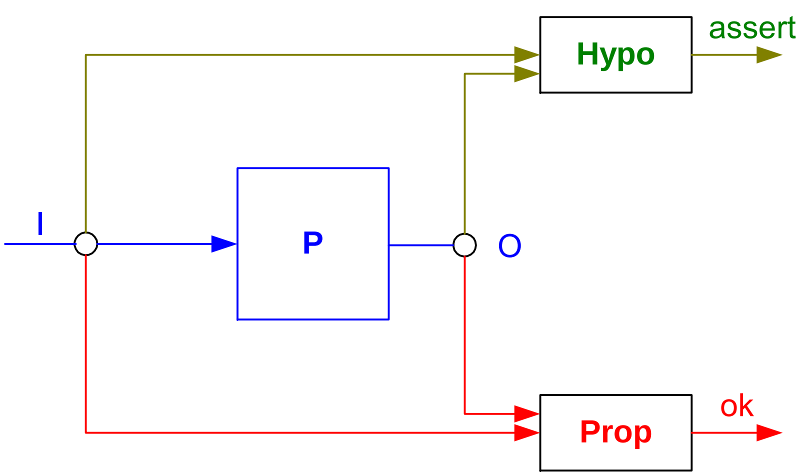



(a3,a2,a1,a0)
(b3,b2,b1,b0)
false
a3 b3
a2 b2
a1 b1
a0 b0
c3
c2
c1
c0
ADD1
ADD1
ADD1
ADD1
s3
s2
s1
s0
carry
(s3,s2,s1,s0)

```
node 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) = ADD1(a0,b0,false);
   (s1,c1) = ADD1(a1,b1,c0);
   (s2,c2) = ADD1(a2,b2,c1);
   (s3,c3) = ADD1(a3,b3,c2);
   carry = c3;
tel;
```

Verification

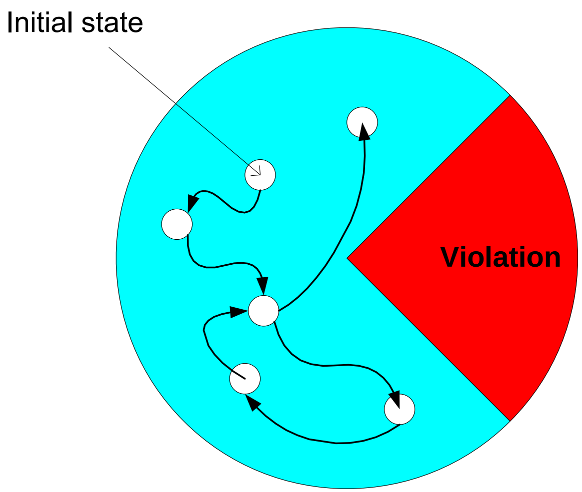
Principle



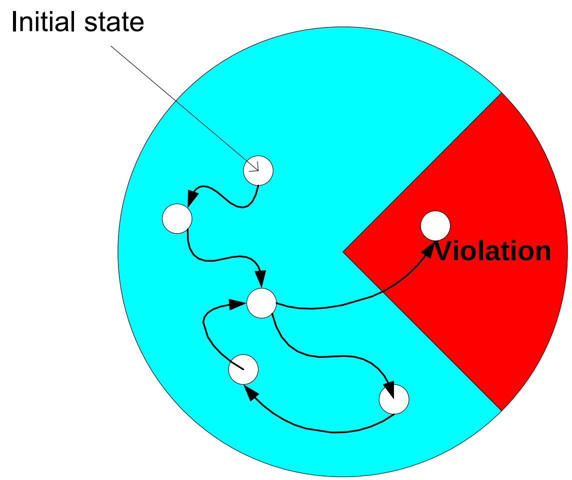

```
node Proof(I)
        returns (ok:bool);
var O;
let
        O=P(I);
        ok=Prop(I,O);
        assert Hypo(I,O);

tel
```

Model-checking

Enumerative algorithm. The automaton is checked state by state, starting from the initial one.



The property holds

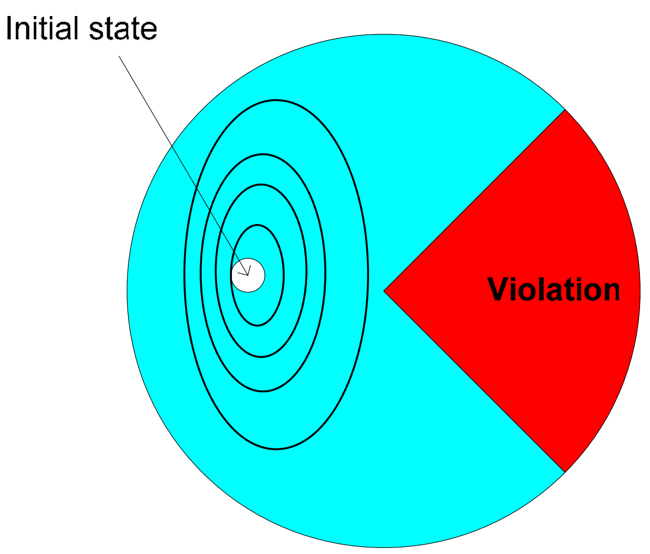


The property is false

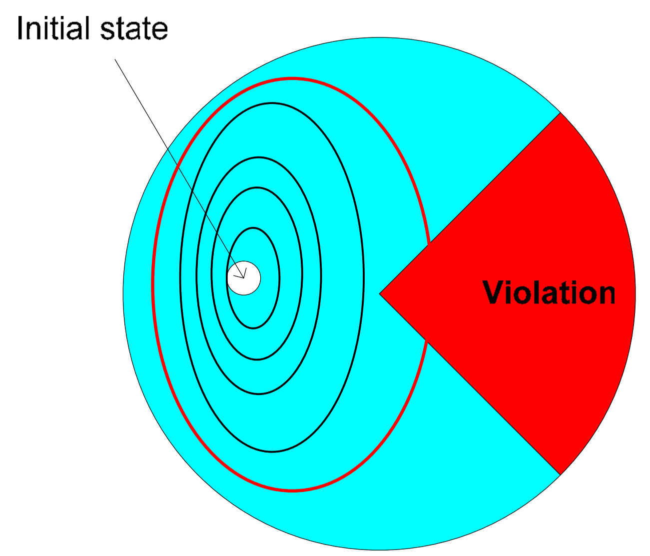
Model-checking (2)

Symbolic forward algorithm: The state of reachable states is built as a Boolean formula over the state variables

`Post*`



The property holds

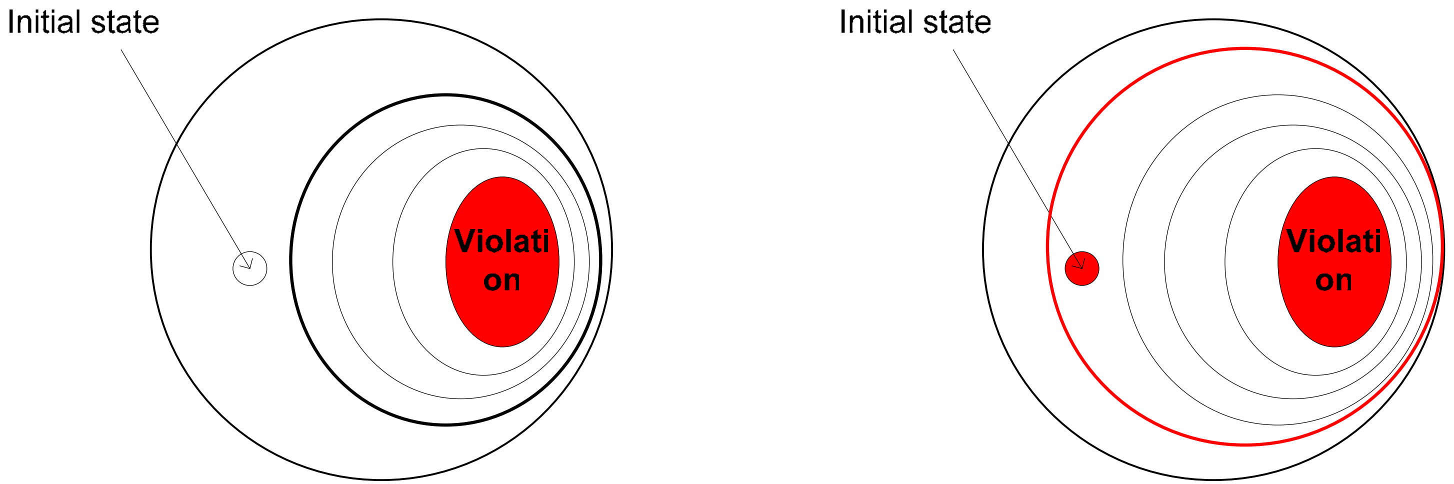


The property is false

Model-checking (3)

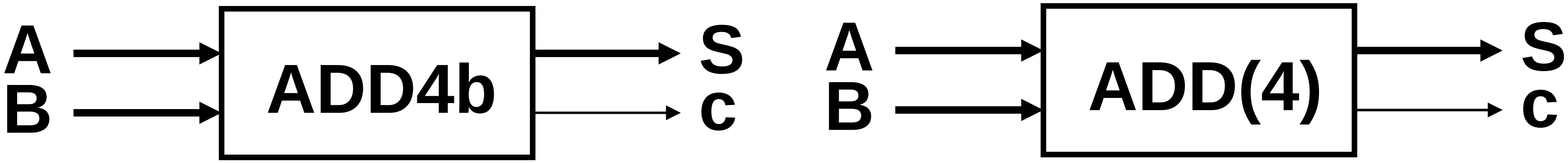
Symbolic backward algorithm: This algorithm builds a symbolic representation of bad states.

Pre*



The property holds

The property is false




```
node verif_ADD (A,B:bool^4)
              returns (ok:bool);
var Sa,Sb:bool^4; ca,cb:bool;
let
   (Sa,ca) = ADD(4,A,B);
   (Sb,cb) = ADD4b(A,B);
   ok = (ca = cb)
     and(Sa[0]=Sb[0])and(Sa[1]=Sb[1])
     and(Sa[2]=Sb[2])and(Sa[3]=Sb[3]);
tel
```

Safety properties

Safety property: bad things never happen.
let P be a safety property.
Define a Boolean expression B such that P is true iff B is `true` through all program executions.

e.g.:

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

```
node neverBefore (B:bool)
                returns (nB:bool);
let
   nB = (not B) -> (not B and pre(nB));
tel
```

k	1	2	3	4	5	6
B	tt	tt	tt	tt	tt	tt
nB	ff	ff	ff	ff	ff	ff

```
node neverBefore (B:bool)
                    returns (nB:bool);
let
  nB = (not B) -> (not B and pre(nB));
tel
```

k	1	2	3	4	5	6
B	ff	ff	ff	tt	tt	tt
nB	tt	tt	tt	ff	ff	ff

```
node since (X,Y:bool)
            returns (XsY:bool);
let
   XsY = if Y then X else
             true -> X or pre(XsY);
tel
```

k	1	2	3	4	5	6
X	ff	ff	ff	tt	ff	tt
Y	ff	ff	tt	ff	ff	tt
XsY	tt	tt	ff	tt	tt	tt

Any occurrence of A is followed by an occurrence of B before the next occurrence of C

future

C A B B C A B C

time

Any time C occurs, either A never occurered previously, or B has occurred since the last occurrence of A

past

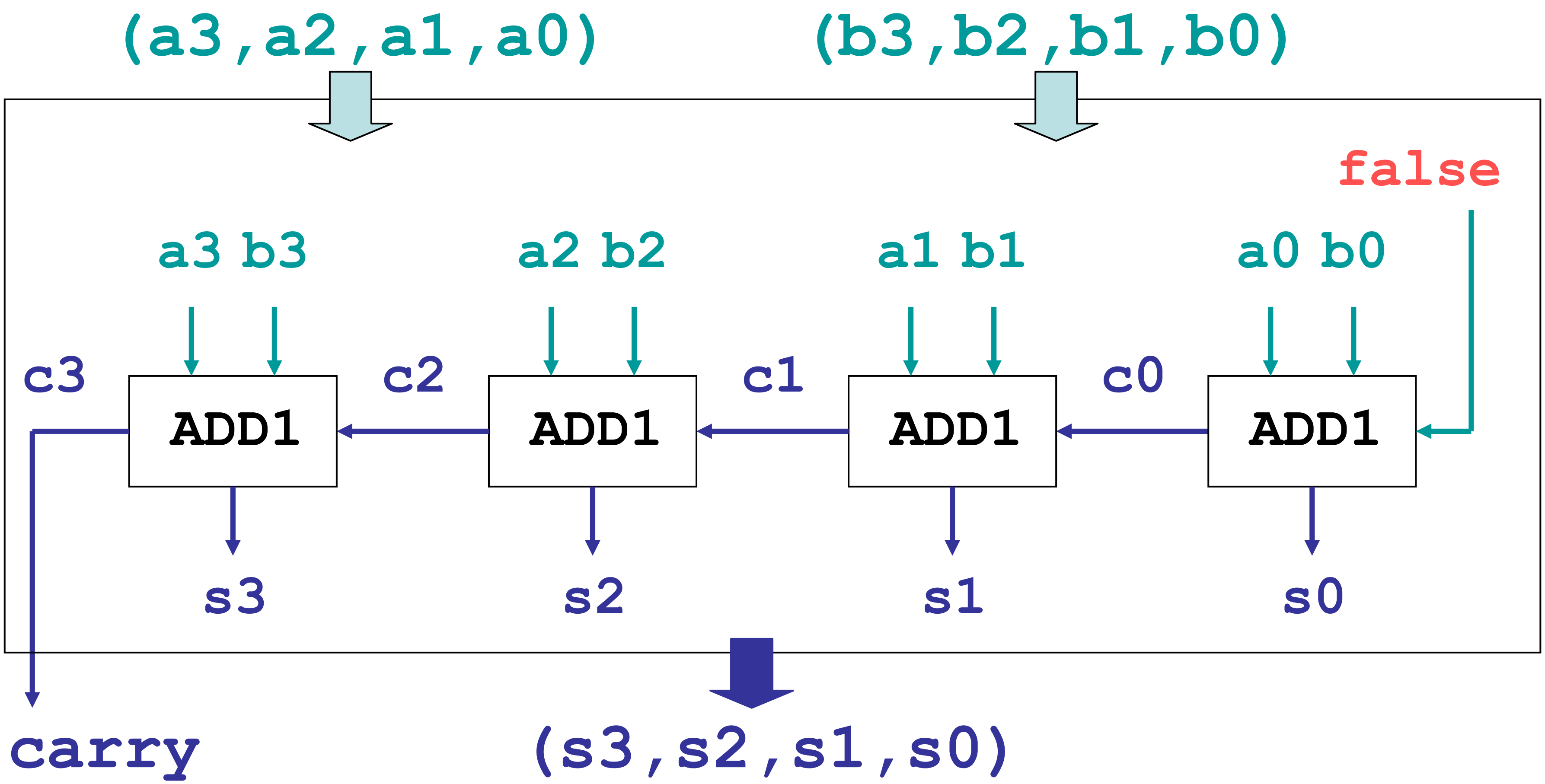
```
node unBentreAetC (A,B,C:bool)
          returns (X:bool);

let
    X = implies(C, neverBefore(A)
        or
        since(B,A));
tel
```

or B has occurred since the last occurrence of A

Any time C occurs, either A has neiver occurred previously,

Lustre

Advanced programming


(a3,a2,a1,a0)
(b3,b2,b1,b0)
false
a3 b3
a2 b2
a1 b1
a0 b0
c3
c2
c1
c0
ADD1
ADD1
ADD1
ADD1
s3
s2
s1
s0
carry
(s3,s2,s1,s0)

```
node 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) = ADD1(a0,b0,false);
    (s1,c1) = ADD1(a1,b1,c0);
    (s2,c2) = ADD1(a2,b2,c1);
    (s3,c3) = ADD1(a3,b3,c2);
    carry = c3;
tel;
```

```
node ADD4a (A,B:bool^4)
     returns (S:bool^4; carry:bool);

 var C:bool^4;
 let
     (S[0],C[0])= ADD1(A[0],B[0],false);
     (S[1..3],C[1..3])= ADD1(
         A[1..3],B[1..3],C[0..2]);
     carry = C[3];
 tel;
```

```
node ADD (const n; A,B:bool^n)
     returns (S:bool^n; carry:bool);
 var C:bool^n;
 let
     (S[0],C[0])= ADD1(A[0],B[0],false);
     (S[1..n-1],C[1..n-1])= ADD1(
          A[1..n-1],B[1..n-1],C[0..n-2]);
     carry = C[n-1];
 tel;
```

```
const nbit = 4;
node Main_ADD (A,B:bool^nbit)
     returns (S:bool^nbit; carry:bool);
 let
     (S,carry)= ADD(nbit,A,B);
 tel;
```

Array operators

- constructor **[.]**

if E0, E1, En-1 are expressions of type T
then **[E0,E1,…,En-1]** is of type **T^n**

- concatenation **|**

let X : T^n and Y: T^m
X | Y is of type **T^(n+m)**

Mutual exclusion

$$EX[i] = \left| \left\{ j \leq i \,\middle|\, X[j] = \text{true} \right\} \right| \leq 1$$

At most 1 true

$$OR[i] = \bigvee_{j \leq i} X[j]$$

At least 1 true

Recurrent equations:

EX[i+1] = EX[i] $\wedge$ ¬ (OR[i] $\wedge$ X[i+1]) where EX[0] = true
OR[i+1] = OR[i] $\vee$ X[i+1] where OR[0] = X[0]

OR[0] = X[0]

OR[1..n-1]

OR = [X[0]] | (OR[0..n-2] or X[1..n-1]);

$OR[i+1] = OR[i] \vee X[i+1]$

EX[0] = true

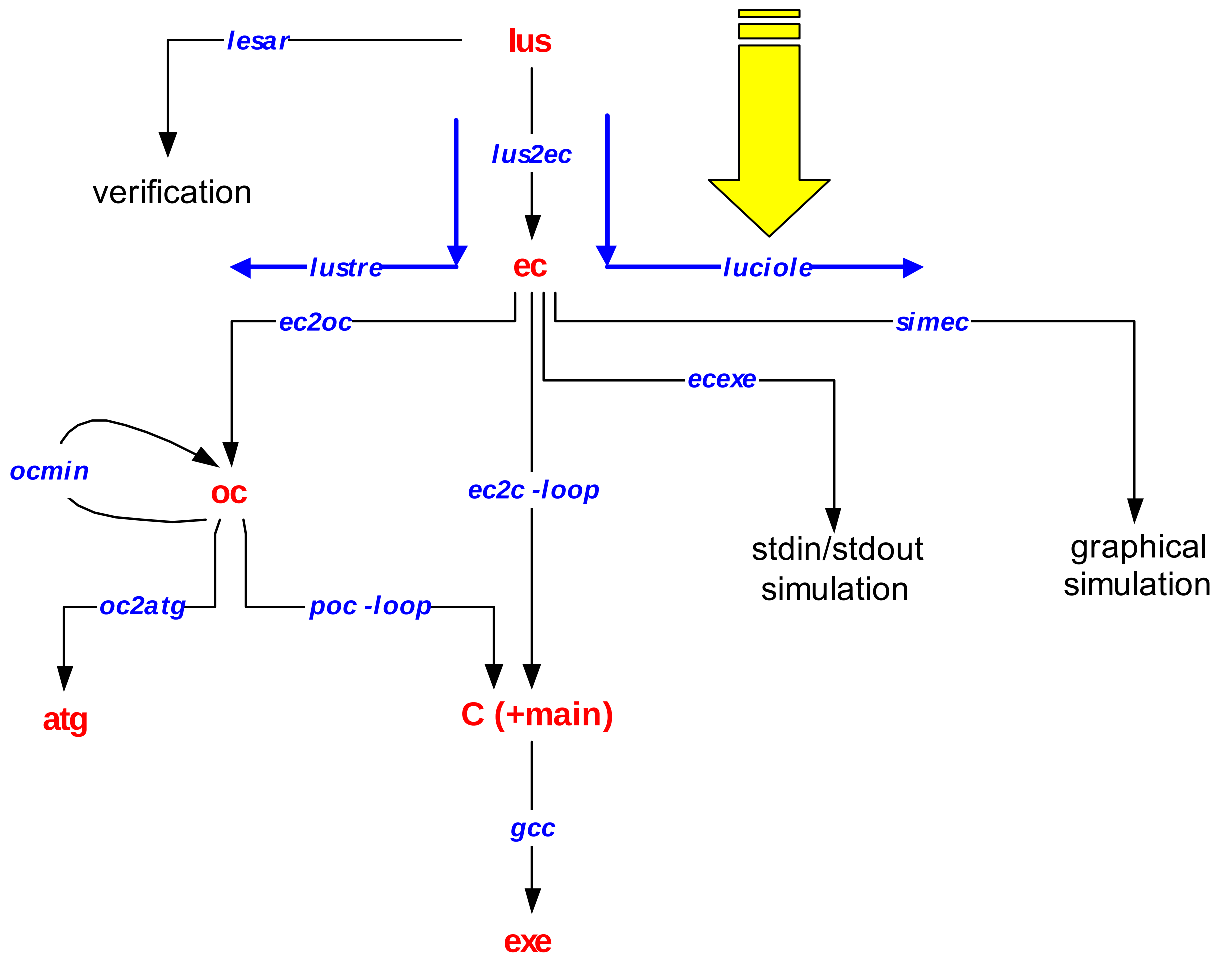
EX[1..n-1]

```
EX = [true] | (EX[0..n-2] and
     not (OR[0..n-2] and X[1..n-1]));
```

$$EX[i+1] = EX[i] \wedge \neg (OR[i] \wedge X[i+1])$$

```
node exclusive (const n:int; X:bool^n)
          returns (excl:bool);
var EX, OR: bool^n;
let
   excl = EX[n-1];
   EX = [true]  | (EX[0..n-2] and
              not (OR[0..n-2] and X[1..n-1]));
   OR = [X[0]] | (OR[0..n-2] or X[1..n-1]);
tel;
```

Tools

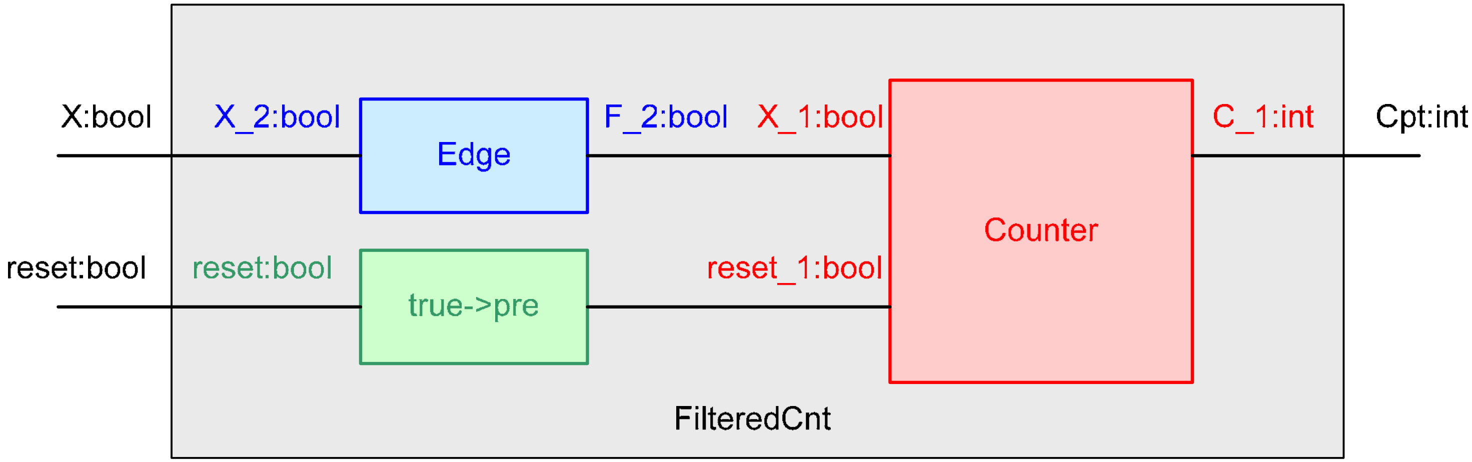


lesar
lus
verification
lus2ec
lustre
ec
luciole
ec2oc
simec
ecexe
ocmin
oc
ec2c -loop
stdin/stdout
simulation
graphical
simulation
oc2atg
poc -loop
atg
C (+main)
gcc
exe

Compilation : source code

```
node Counter(X, reset:bool) returns (C:int);
let
        C =     if reset then 0
                else    if X then (0 -> pre C) + 1
                        else (0 -> pre C) ;
tel
node Edge(X:bool) returns (F:bool);
let
        F = X -> X and not pre X;
tel

node FilteredCnt(X, reset:bool) returns (cpt:int);
let
        cpt = Counter(Edge(X), (true -> pre reset));
tel
```

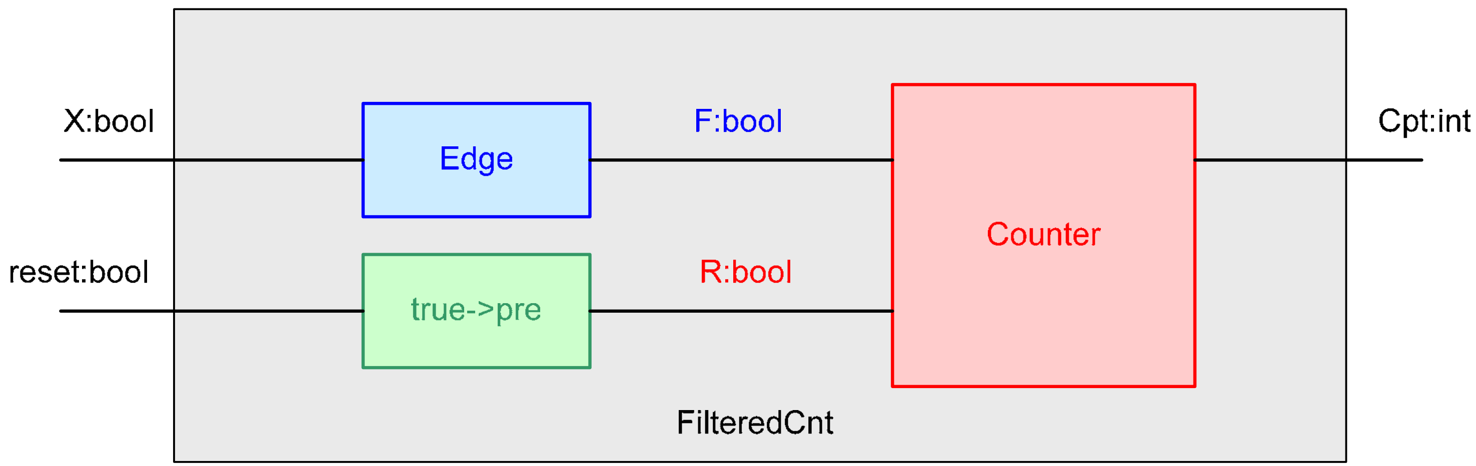
Compilation : expansion (1)



Compilation : expansion (2)

```
node FilteredCnt(X, reset:bool) returns (cpt:int);
var
        X_1, reset_1:bool; -- inputs of Counter
        C_1:int; -- output of Counter
        X_2, F_2:bool; -- input and output of Edge
let
        -- Calling Counter:
        C_1 =   if reset_1 then 0
                else    if X_1 then (0 -> pre C_1) + 1
                        else (0 -> pre C_1) ;
        X_1 = F_2;
        reset_1 = true -> pre reset;
        cpt = C_1;
        -- calling Edge:
        F_2 = X_2 -> X_2 and not pre X_2;
        X_2 = X;
tel
```

Compilation : expansion (3)



Compilation : expansion(4)

```
node FilteredCnt(X, reset:bool) returns (cpt:int);
var
        F, R:bool;
let
        cpt =  if R then 0
               else   if F then (0 -> pre cpt) + 1
                      else (0 -> pre cpt) ;
        R = true -> pre reset;
        F = X -> X and not pre X;
tel
```

Compilation: to imperative code

memories

```
where
          pcpt stands for « pre cpt »

          pX stands for  « pre X »

          preset stands for « pre reset »

cpt =     if R then 0
          else
                    if F then (if init then 0 else pcpt) + 1
                    else (if init then 0 else pcpt) ;
R = if init then true else preset;
F = if init then X else (X and not pX);
```

assignements

Compilation : sequentialization

```
R = if init then true else preset;
F = if init then X else (X and not pX);
cpt = if R then 0
      else  if F then (if init then 0 else pcpt) + 1
            else (if init then 0 else pcpt);
```

Ordered (assignments)

Compilation : C code (1)

```
/* memories are global because their values are persistent
from a cycle to the next one */
static int pcpt;
static bool pX;
static bool preset;
static bool init = true; /* Initialization */

/* Function that implements a reaction */
void FilteredCnt_step (
         bool X; bool reset; /* inputs passed by value */
         int* cpt /* output passed by reference */
         )
```

Compilation : C code (2)

```
{
        bool R, F; /* local variables. Non persistent values */
        /* Local variables and output computation */
        R = init? true : preset;
        F = init? X : (X && ! pX);
        *cpt = R? 0 :
                 F? ((init? 0 : pcpt) + 1) :
                        init? 0 : pcpt;
        /* Memorizing */
        pcpt = *cpt;
        pX = X;
        preset = reset;
        init = false;
}
```

Compilation : C code (3)

```
{
        bool F; /* local variables. Non persistent values */
        if (init) {
                /* simplified code when init = true; F useless */
                *cpt = 0;
                /* assignment executed once only */
                init = false;
        } else {
                F = (X && ! pX);
                *cpt = preset? 0 :
                        F? (pcpt + 1): pcpt;
        }
        /* Memorizing */
        pcpt = *cpt;
        pX = X;
        preset = reset;
}
```

imperative
if then else

Compilation (format)

Single loop

- size of the code linear w.r.t. the size of the source code
- Systematic execution of the body: may be slow!

Automaton

- Size of the code can be exponential
- Execution: depends on the current state (switch): fast

SIGNAL

Signal

- A **signal** is a sequence of values associated with a **clock**.
- Data:
  - Scalar types (`boolean, integer, float`)
  - Arbitrary dimension array of scalar elements
  - type `event` , which may be present or absent
- A clock is a set of discrete instants out of a totally ordered set.

Let $\mathbf{X}$ be a signal. The clock $\mathbf{C_X}$ associated with $\mathbf{X}$ is defined by the sequence of instants when $\mathbf{X}$ is present.

$$X = \left(X_t \right)_{t \in C_X}$$

Given a clock $\mathbf{C}$, one defines

$$0_C = \min \left\{ t \,\middle|\, t \in C \right\}$$

$$\forall t \in C, t \neq 0_C, t <<_C 1 = \max \left\{ t' \in C, t' < t \right\}$$

$$t <<_C k + 1 = \max \left\{ t' \in C, t' <<_C k \right\} \text{ si } t <<_C k \neq 0_C$$

Operators

Signals are defined by elementary processes built with two kinds of operators:

- Usual operators extented to sequences

$\boxed{Y := f(X1, \ldots, Xn)}$ Y,X1, …, Xn same clock

- Temporal operators:
  - delay Y := X $ k

$$\forall t \in C \,\middle|\, t <<_C k \,, \exists Y_t = X_{t <<_C k}$$

  - extraction Y:=X when B (B Boolean signal)

$$C_Y = \left\{ t \in C_X \cap C_B \,\middle|\, B_t = \text{true} \right\} \text{ et } \forall t \in C_Y, Y_t = X_t$$

- deterministic merge

$\boxed{\text{Y := X default Z}}$ $\quad C_Y = C_X \cup C_Z$

$$\forall t \in C_X \cup C_Z, \quad Y_t = \begin{cases} X_t & \text{if} \quad t \in C_X \\ Z_t & \text{otherwise} \end{cases}$$

Consequence: Clocks in Signal are not trees (differs from Lustre)

Composition

Two composition operators:

- parallel composition

P | Q

Conjunction of constraints

- restriction

P \ X

processus signal

makes X local to P

+ synchro operator

Example of Signal program

The signal **MIN** must be emitted every 60 **SEC**

```
(| S := (0 when MIN) default (ZS+1)
 | ZS := S $ 1
 | MIN := SEC when (ZS=59)
 | synchro {S,SEC}
 |)
```

Clock calculus

- Relation between clocks are encoded as polynomial over the field $\mathbb{Z}/3\mathbb{Z}$
- Absent $\leftrightarrow$ 0
- Boolean present **`true`** $\leftrightarrow$ 1
- Boolean present **`false`** $\leftrightarrow$ -1

Clocks (continued)

Given a Boolean flow s, $-s-s^2$= 1 iff s is present and **`true`**.

`v when s` $\Rightarrow$ product of the polynomial of v by $-s-s^2$

The polynomial $1-s^2$ is 0 iff s is present.

`s default v` $\Rightarrow$ $s+(1-s^2)v$

For non Boolean flows, all present values are represented by 1, all absent values by 0.