Langages synchrones déclaratifs

Lustre / Signal

Langages

On dit ce qui EST ou qui DOIT ETRE

Langages déclaratifs

Langages impératifs

On dit ce qu'on DOIT FAIRE

LUSTRE/SIGNAL

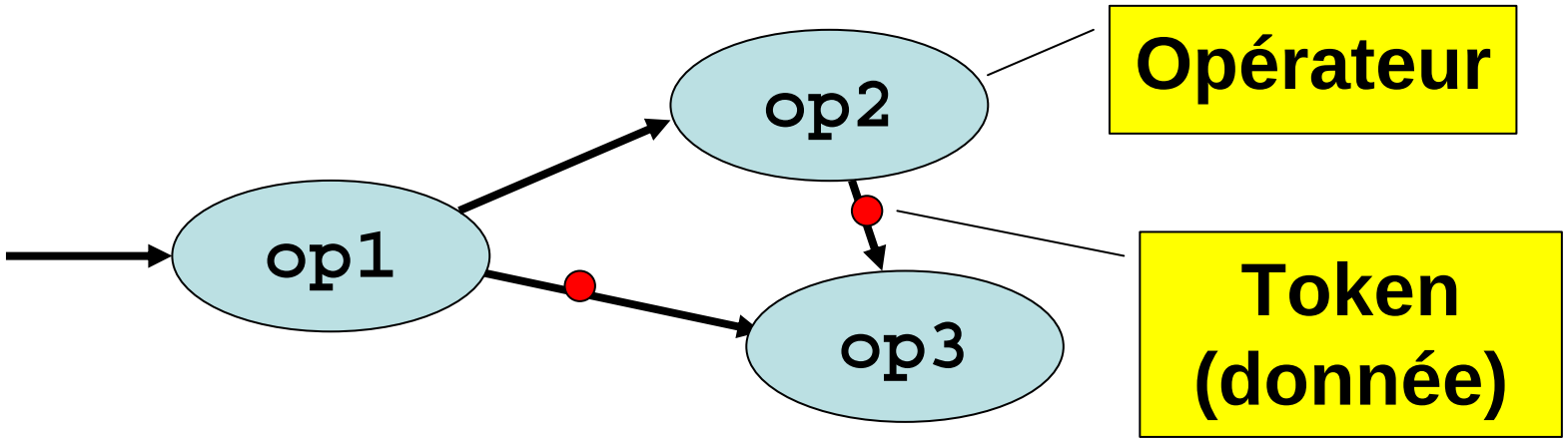
Langage	Caractéristiques	Applications
LUSTRE (et l'outil SCADE associé)	Fonctionnel Mono horloge	Systèmes critiques; Commande; Circuits
SIGNAL	Relationnel (contraintes) Multi horloges	Spécification de systèmes. Automatique; Traitement du signal

LUSTRE

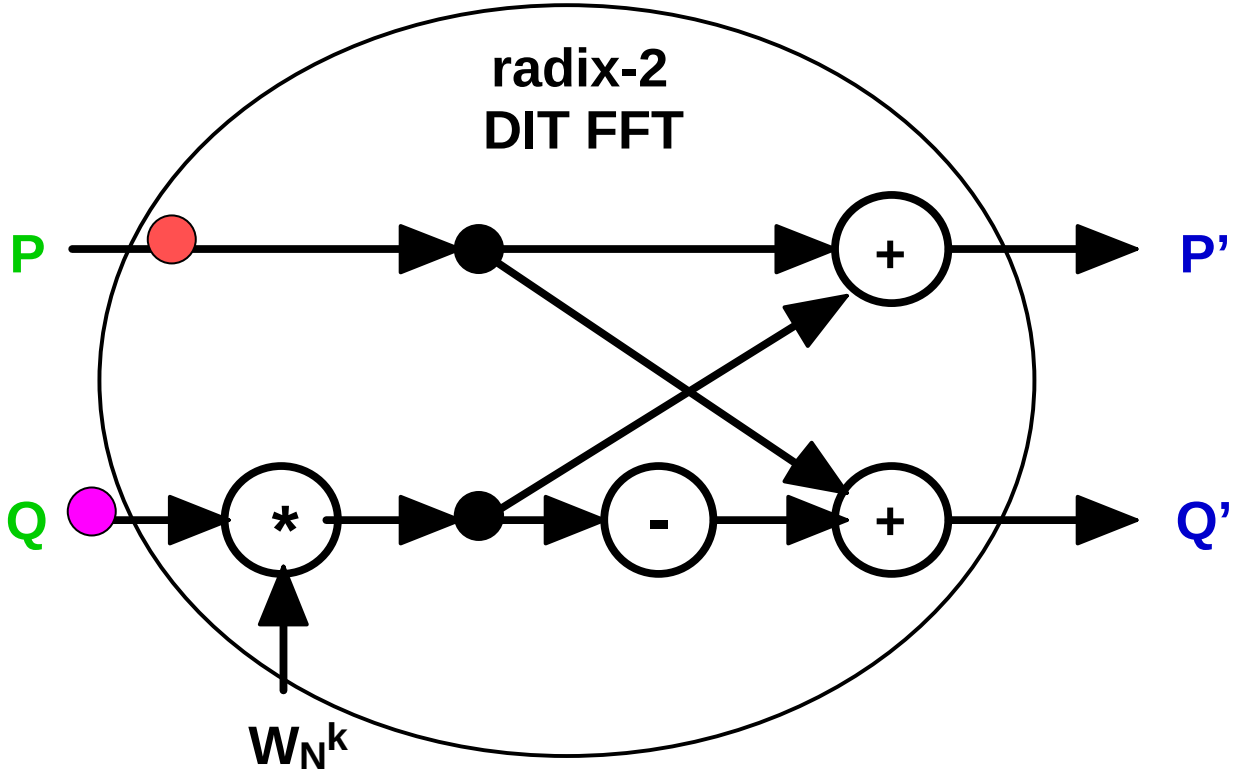
- Nous allons étudier LUSTRE en détail car
  - C'est un langage très simple (4 opérateurs primitifs pour exprimer les réactions)
  - Basé sur des modèles familiers pour les ingénieurs:
    - Systèmes d'équations
    - Réseaux flot de données
  - Se prête bien à la vérification (langage logique par certains aspects)
  - Sémantique très simple

Réseaux d'opérateurs

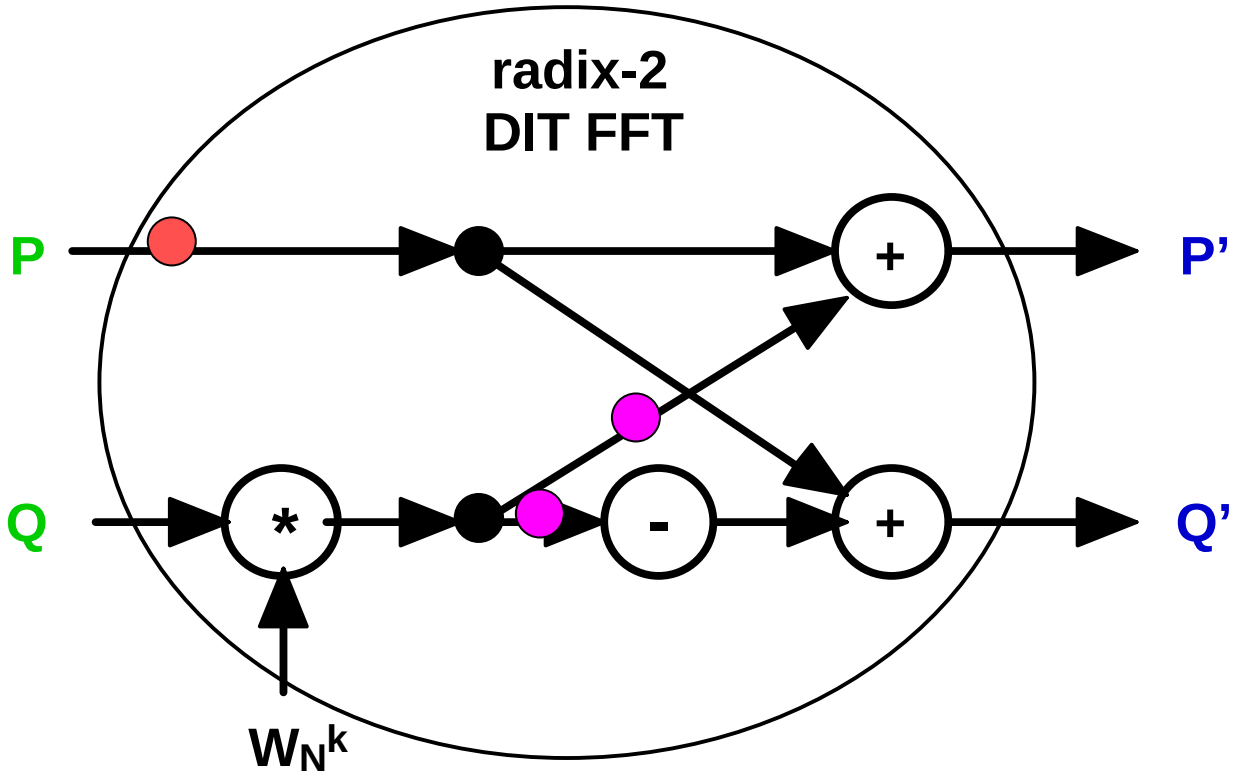
- En LUSTRE et SIGNAL les programmes peuvent être interprétés comme des réseaux d'opérateurs.
- Les données « vont » vers les opérateurs où elles sont consommées. Ces opérateurs à leur tour engendrent de nouvelles données. (Description « flot de données » ou Data Flow).



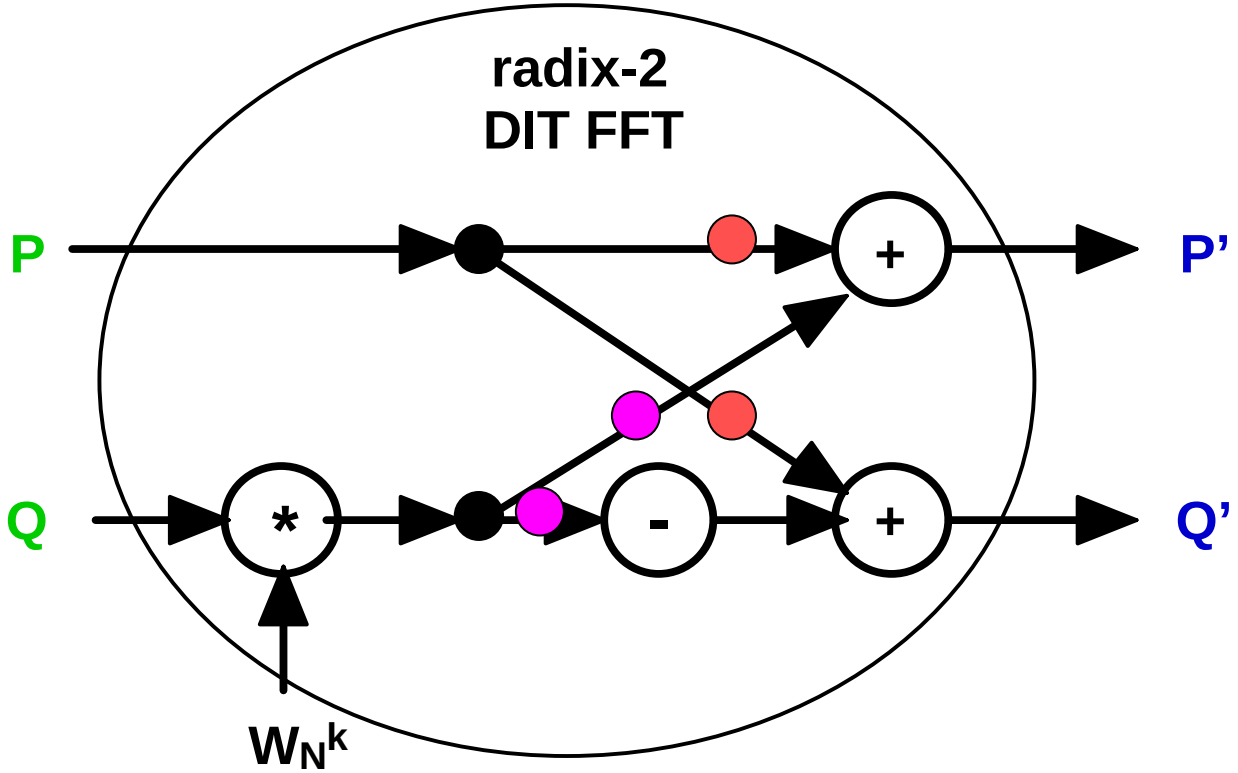
Exemple de Flots de données



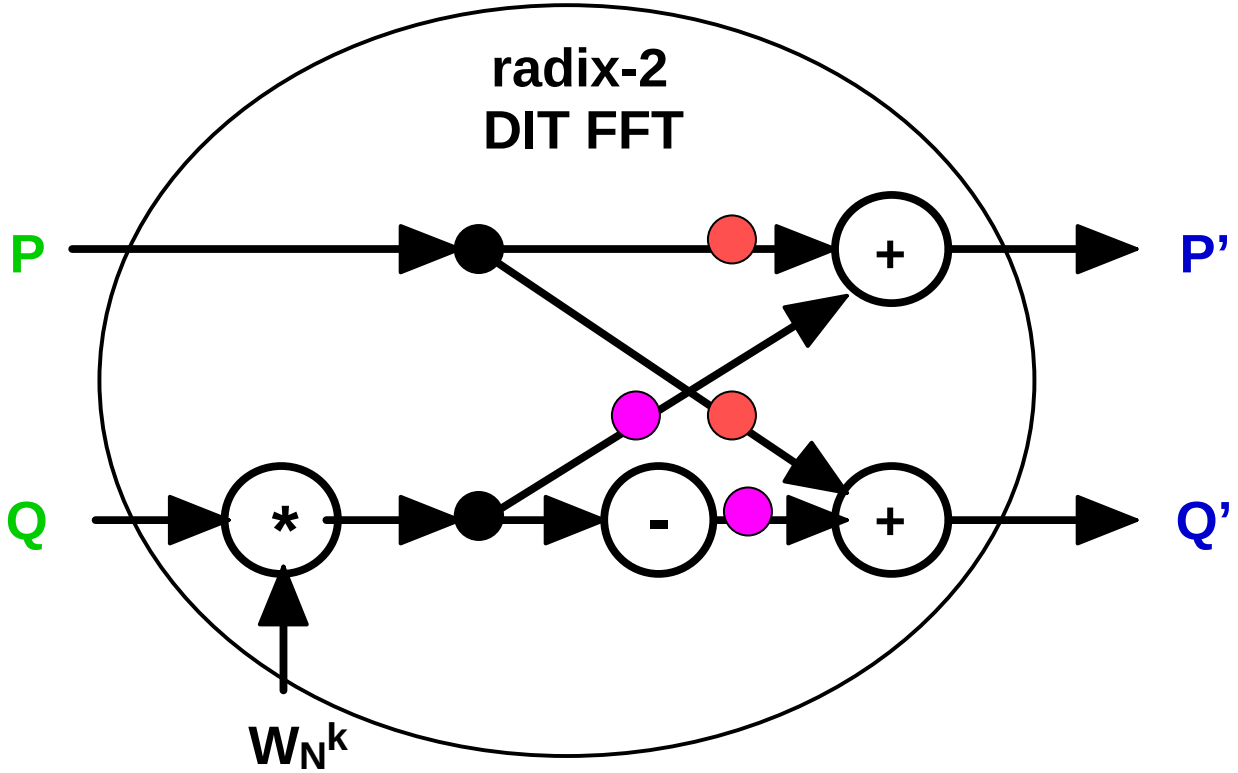
Flots de données



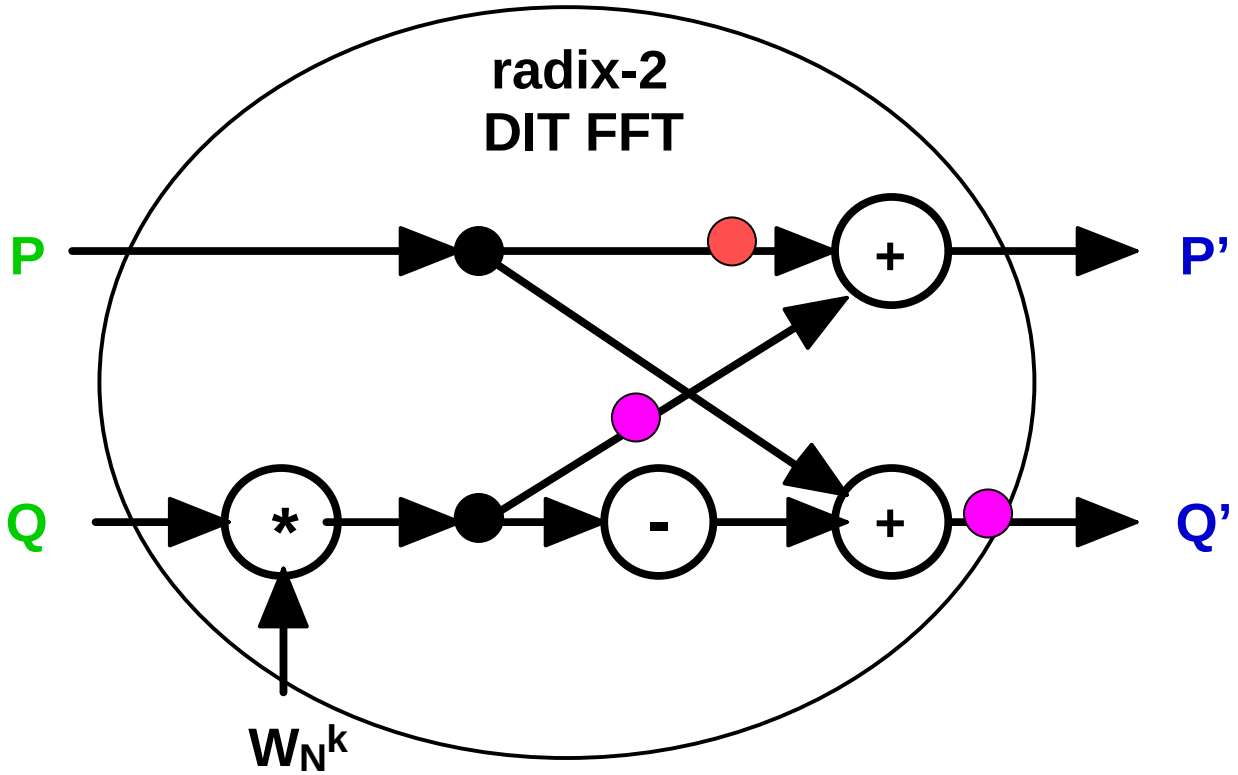
Flots de données



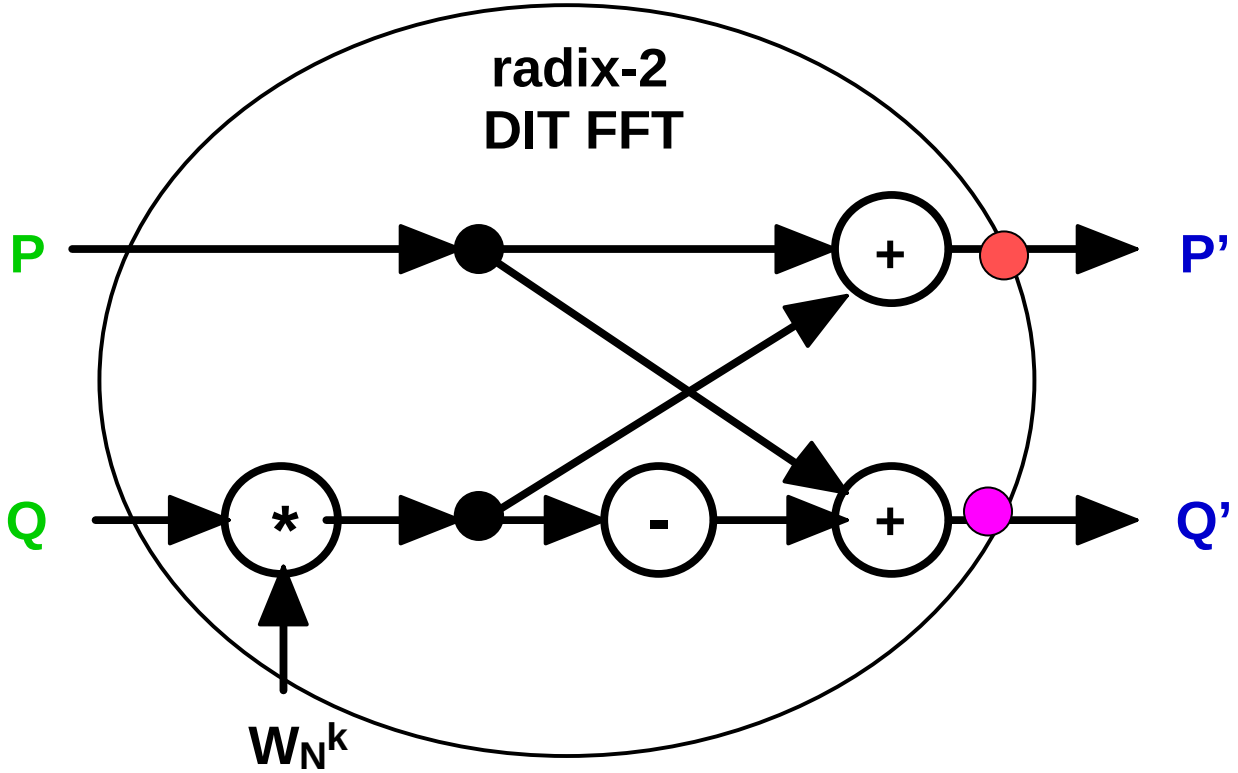
Flots de données



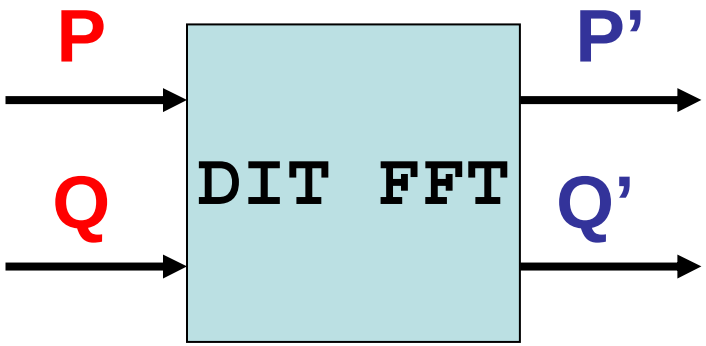
Flots de données



Flots de données



Point de vue fonctionnel



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

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

LUSTRE

Flots, horloges

- Un flot est une paire constituée de
  - une séquence de valeurs typées
  - une horloge qui représente une séquence d'instants

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

Langage (1)

Variable :

- typée
- si pas entrée, définie par 1 et 1 seule équation
- types prédéfinis: **`int`**, **`bool`**, **`real`**
- uplets: **`(a,b,c)`**

Equation :

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

Assertion :

Expression booléenne supposée **`true`** à chaque instant de son horloge.

Langage (2)

Principe de substitution :

si `X = E` alors `E` peut être substitué à `X` n'importe où dans le programme, et réciproquement

Principe de définition :

une variable est complètement définie par sa déclaration et l'équation dans laquelle elle apparaît en membre gauche

Expressions

Constantes

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

`int` `bool` `real`

\+
types et opérateurs importés

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

Types d'opérateurs

- Opérateurs stricts :

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

- Opérateurs surchargés :

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

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

- Opérateurs polymorphes :

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

Lustre « combinatoire »

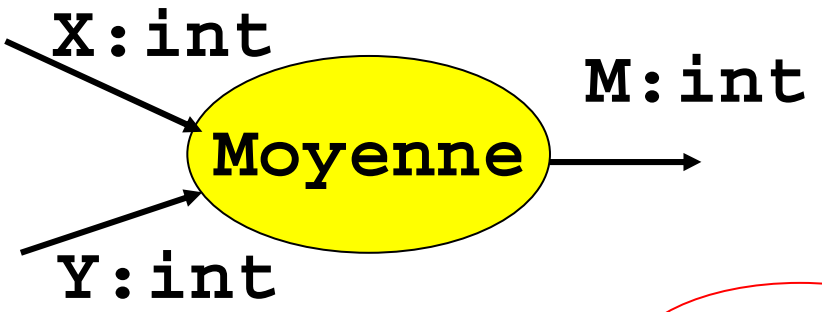
Opérateurs de données

Arithmétiques : **`+`**, **`-`**, **`*`**, **`/`**, **`div`**, **`mod`**
Logiques : **`and`**, **`or`**, **`not`**, **`xor`**, **`=>`**
Conditionnels : **`if … then … else …`**
Casts: **`int`**, **`real`**

Opérations « point à point »

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

Exemple « Combinatoire »



Opérateur

Données

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

Résultat

Définition

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

Exemple (suite)

```
node Moyenne (X,Y:int)
  returns (M:int);
var S:int;   -- variable locale
let
  S = X + Y; -- ordre non significatif
  M = S / 2;
tel
```

Donne le même comportement par substitution

Mémorisation

On tient compte du passé

`pre` (précédent) :

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

Problème des valeurs non définies : nil

`->` (suivi de) :

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

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

Exemples « séquentiels »

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

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

Exo: Modifier pour front initial

uplet

```
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
```

Définitions récursives

Récursions temporelles

Courantes. Utilisent **`pre`** et **`->`**

Ex: `nat = 1 -> pre nat + 1`

Récursions instantanées

Ex: `X = 1.0 / (2.0 - X)`

Interdites en Lustre, même si il existe une solution unique.

Attention aux récursions croisées.

Horloges

Horloge de base

- Temps discret induit par la séquence d'entrée du programme
- Horloges dérivées (plus lentes)

when (opérateur de filtrage) :

E when C est la sous-suite de **E** en ne retenant que les e_k pour lesquels c_k=**true**

Exemples d'horloge

Cycles de base	1	2	3	4	5	6	7	8
C1	true	false	true	true	false	true	false	true
Cycles de C1	1		2	3		4		5
C2	false		true	false		true		true
Cycles de C2			1			2		3

Exemple d'échantillonnage

Pour

```
nat,odd:int

halfBaseClock:bool

nat = 1 -> pre nat +1;

halfBaseClock =

  true -> not pre halfBaseClock;

odd = nat when halfBaseClock;
```

nat est un flot sur l'horloge de base;
odd est un flot sur l'horloge **halfBaseClock**

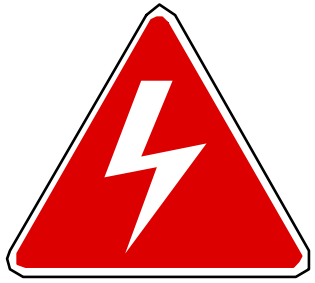
Exo: écrire even

Opérateur de projection

« inverse » de l'échantillonnage

`current` (opérateur de projection) :

`E` étant une expression sur une horloge **`C`**, **`current(E)`** est une expression sur l'horloge de **`C`**, avec maintien de la valeur prise par **`E`** dans le dernier cycle où **`c`** était **`true`**.



`current (X when C) ≠ X`

`current` peut donner des **`nil`**

Exemple de current

Cycles de base	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

Autre exemple de 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							
H=Y when C							
T=Z when H							
current T							
current (current T)							

Autre exemple de 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=Z when H							
current T							
current (current T)							

Autre exemple de 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							
current T							
current (current T)							

Autre exemple de 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							
current (current T)							

Autre exemple de 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)							

Autre exemple de 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

Problème de l'initialisation

`Y=current(X when C)` avec C_1 = false donne une erreur.

Solutions possibles:

- Discipline d'utilisation: faire en sorte que C_1 soit toujours true.
- Forcer l'horloge à true au premier instant:

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

- Donner une valeur par défaut D :

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

```
Y = if C then X else D -> pre Y;
```

Premiers programmes

Compteur

```
node COMPTEUR (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
COMPTEUR(0,2,false)	0	2	4	6	8	10	12
COMPTEUR((0,2,false)when C)	0		2	4			6
COMPTEUR(0,2,false)when C	0		4	6			12

Front

```
node Front (b:bool) returns (f:bool);
-- détection de front montant
let
    f = false -> (b and not pre(b));
tel;
```

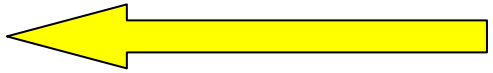
initial

Non défini initialement

```
Front_Desc = Front(not c);
```

Monostable réarmable

```
node MR(set:bool;delay:int) returns (z:bool);
-- monostable réarmable
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;
```



Variable locale

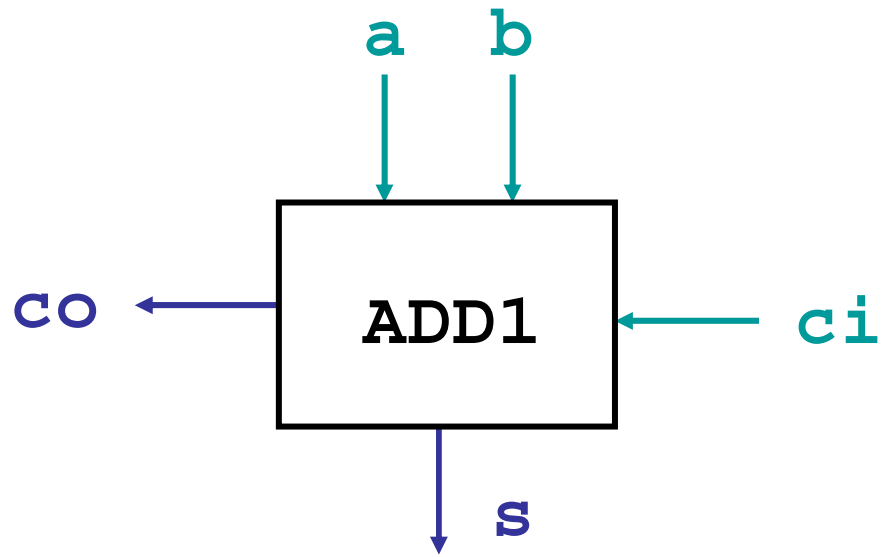
2 équations: ordre **non significatif**

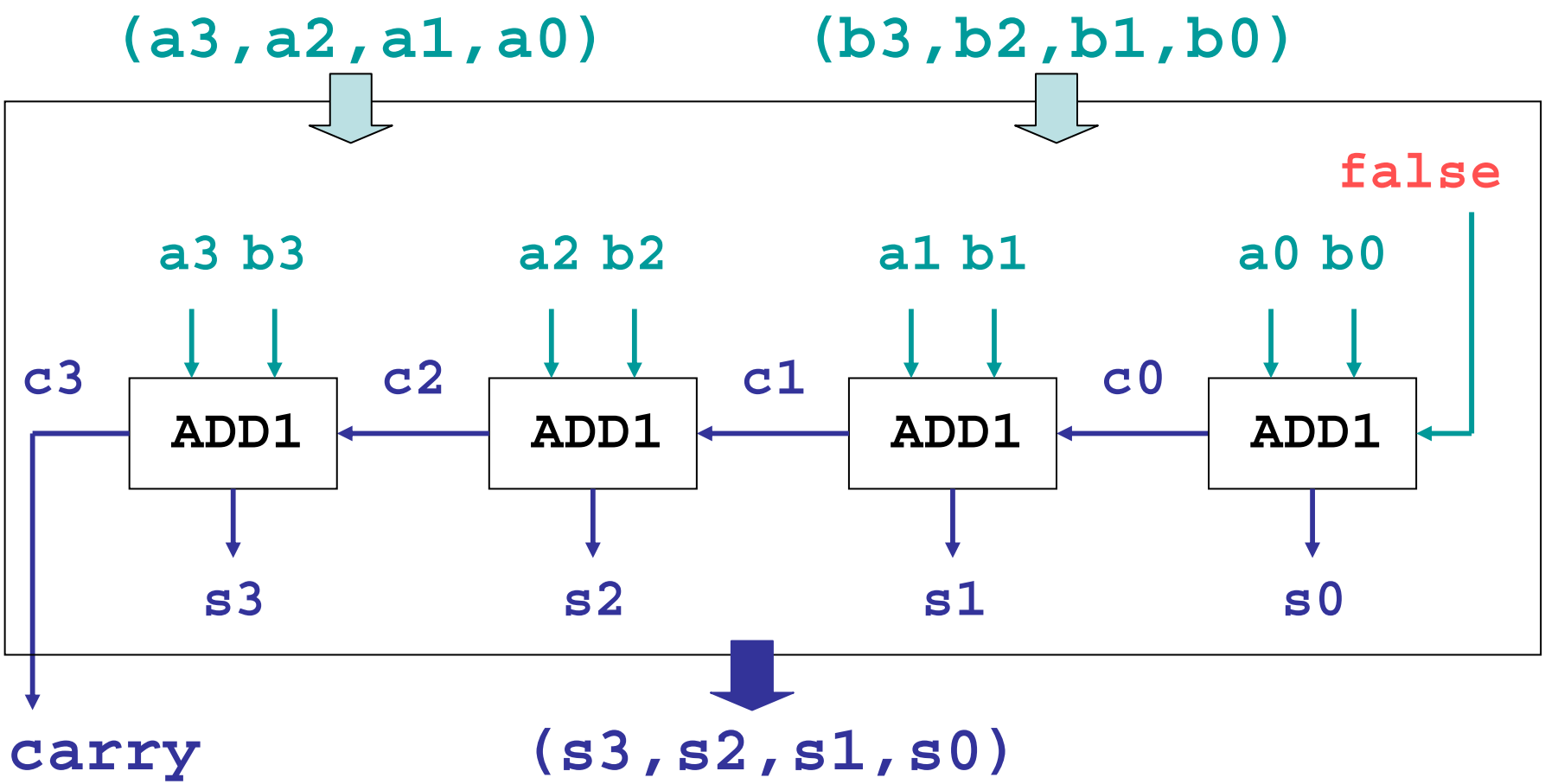
Monostable non réarmable

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

Additionneur

```
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
```

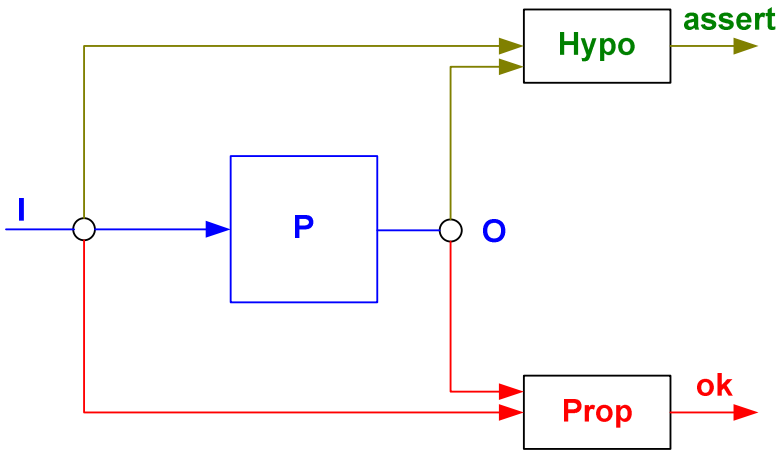



(a3,a2,a1,a0)
(b3,b2,b1,b0)
false
a3 b3
a2 b2
a1 b1
a0 b0
c3
c2
c1
c0
ADD1
ADD1
ADD1
ADD1
s3
s2
s1
s0
carry
(s3,s2,s1,s0)

```
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;
```

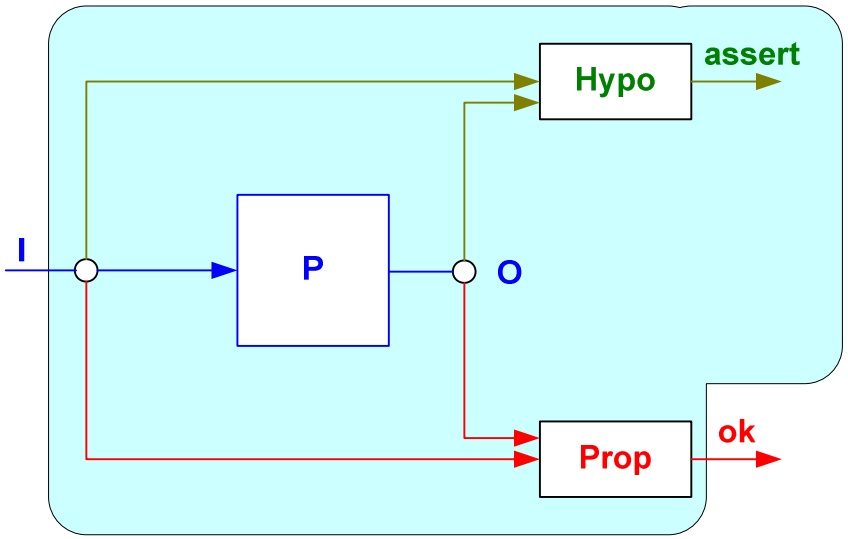
Vérification

Principe de la vérification




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

tel
```

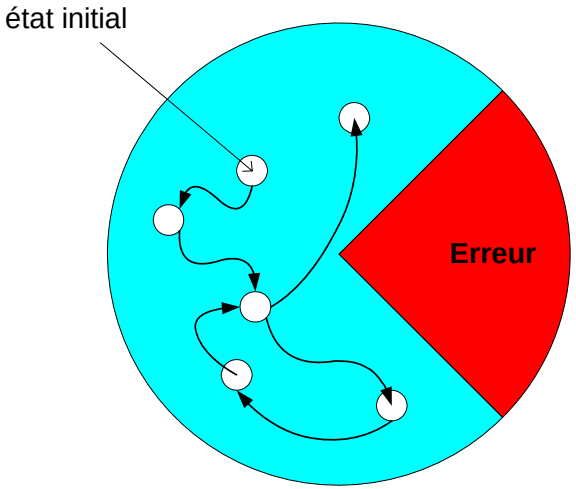
Principe de la vérification

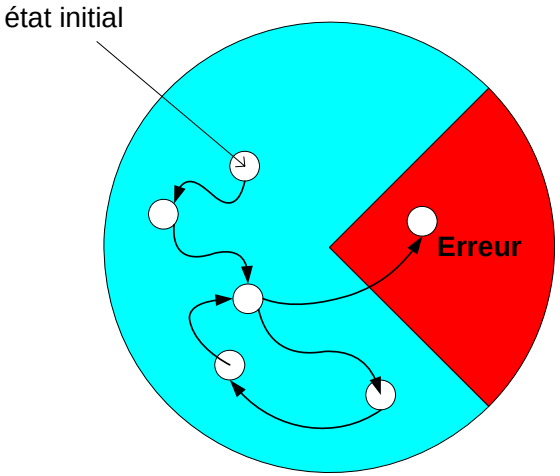



```
node Preuve(I)
        returns (ok:bool);
var O;
let
        O=P(I);
        ok=Prop(I,O);
        assert Hypo(I,O);
tel
```

Model-checking

Parcours énumératif de l'automate





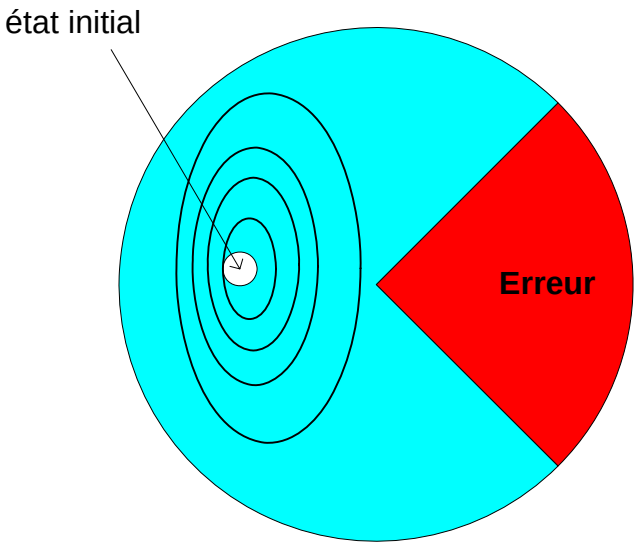
La preuve réussit

La preuve échoue

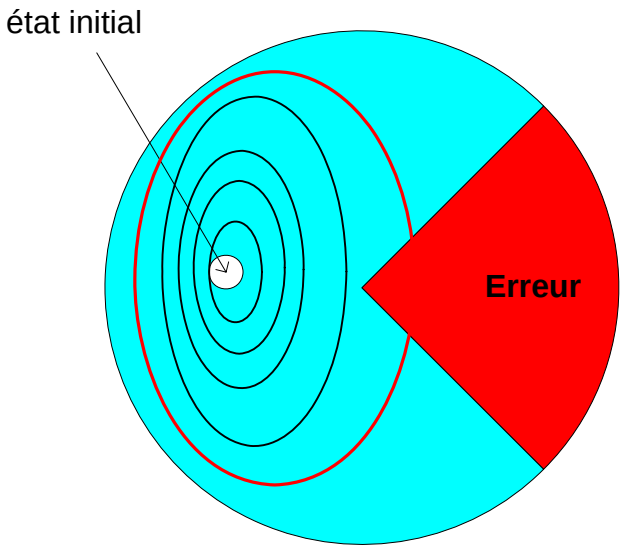
Model-checking (2)

Méthode symbolique : parcours en avant

`Post*`



La preuve réussit

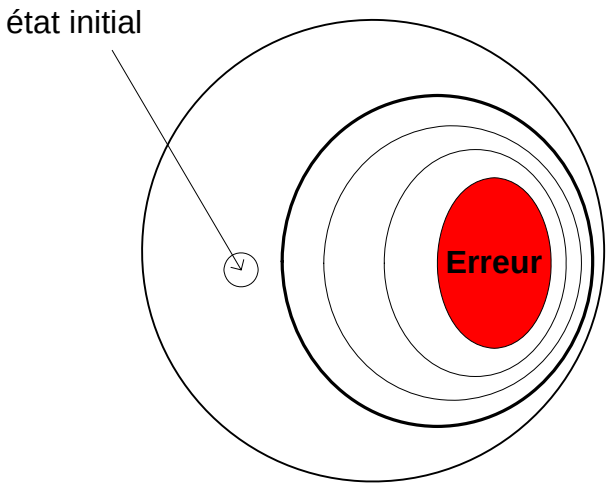


La preuve échoue

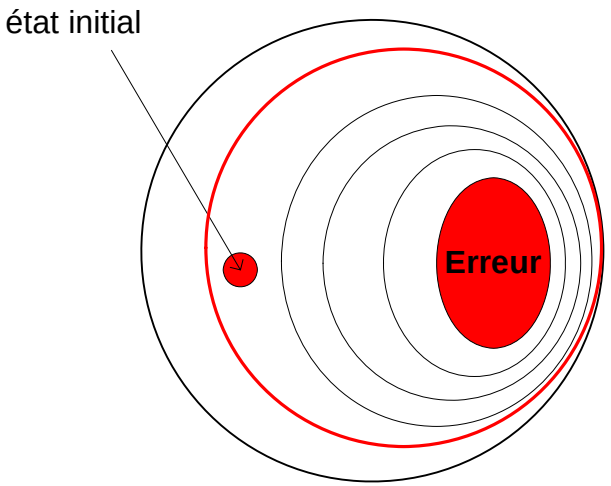
Model-checking (3)

Méthode symbolique : parcours en arrière

`Pre*`



La preuve réussit



La preuve échoue

Propriétés de sûreté (safety)

Pour tester une propriété de sûreté P,
On définit une expression booléenne B
telle que
P est vérifiée ssi B reste à `true` durant toute exécution du programme.

Ex:

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

Il n'y a pas encore eu B

```
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

Il y a eu au moins 1 X depuis le dernier Y

```
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

Toute occurrence de A est suivie d'une occurrence de B au moins avant l'occurrence suivante de C

futur

Chaque fois que C se réalise, soit A ne s'est jamais produit auparavant,
Soit B s'est réalisé au moins une fois depuis la dernière occurrence de A

passé

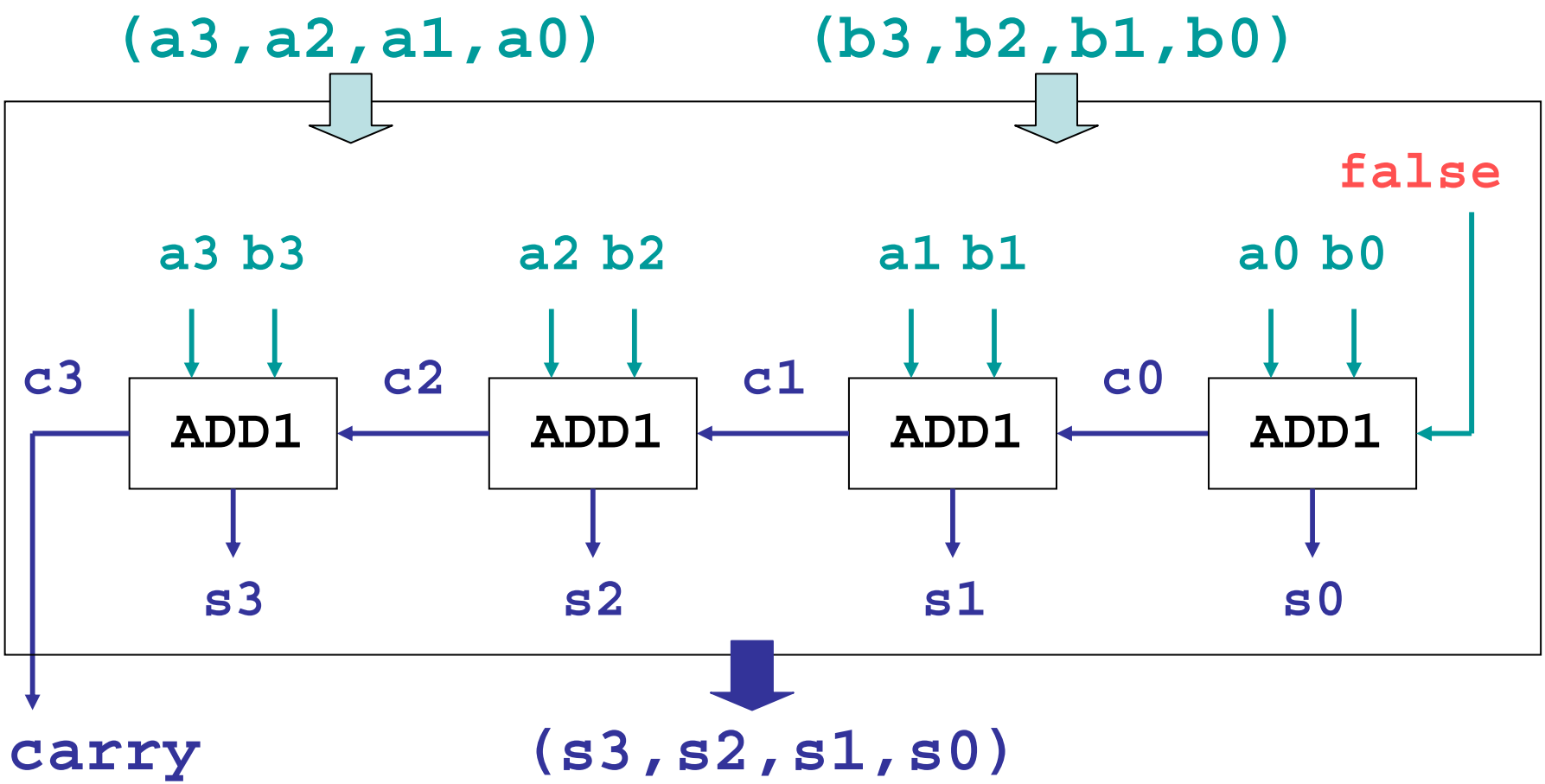
```
node unBentreAetC (A,B,C:bool)
          returns (X:bool);

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

Soit B s'est réalisé au moins une fois depuis la dernière occurrence de A

Chaque fois que C se réalise, soit A ne s'est jamais produit auparavant,

Lustre

Programmation avancée

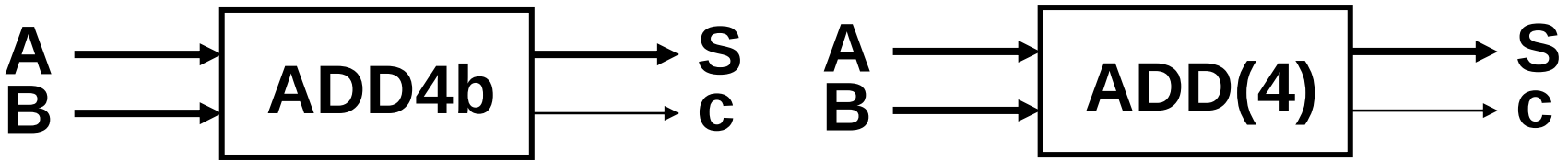

(a3,a2,a1,a0)
(b3,b2,b1,b0)
false
a3 b3
a2 b2
a1 b1
a0 b0
c3
c2
c1
c0
ADD1
ADD1
ADD1
ADD1
s3
s2
s1
s0
carry
(s3,s2,s1,s0)

```
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;
```

Retour sur la Vérification




```
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
```

Opérateurs sur tableaux

- Le constructeur **[.]**

si E0, E1, En-1 sont des expressions de type T
alors **[E0,E1,…,En-1]** est du type **T^n**

- La concaténation **|**

soit X : T^n et Y: T^m
X | Y est du type **T^(n+m)**

Exclusion mutuelle

$$EX[i] = \left| \{ \; j \le i \; | \; X[j] = \text{true} \; \} \right| \le 1$$

Au plus 1 à true

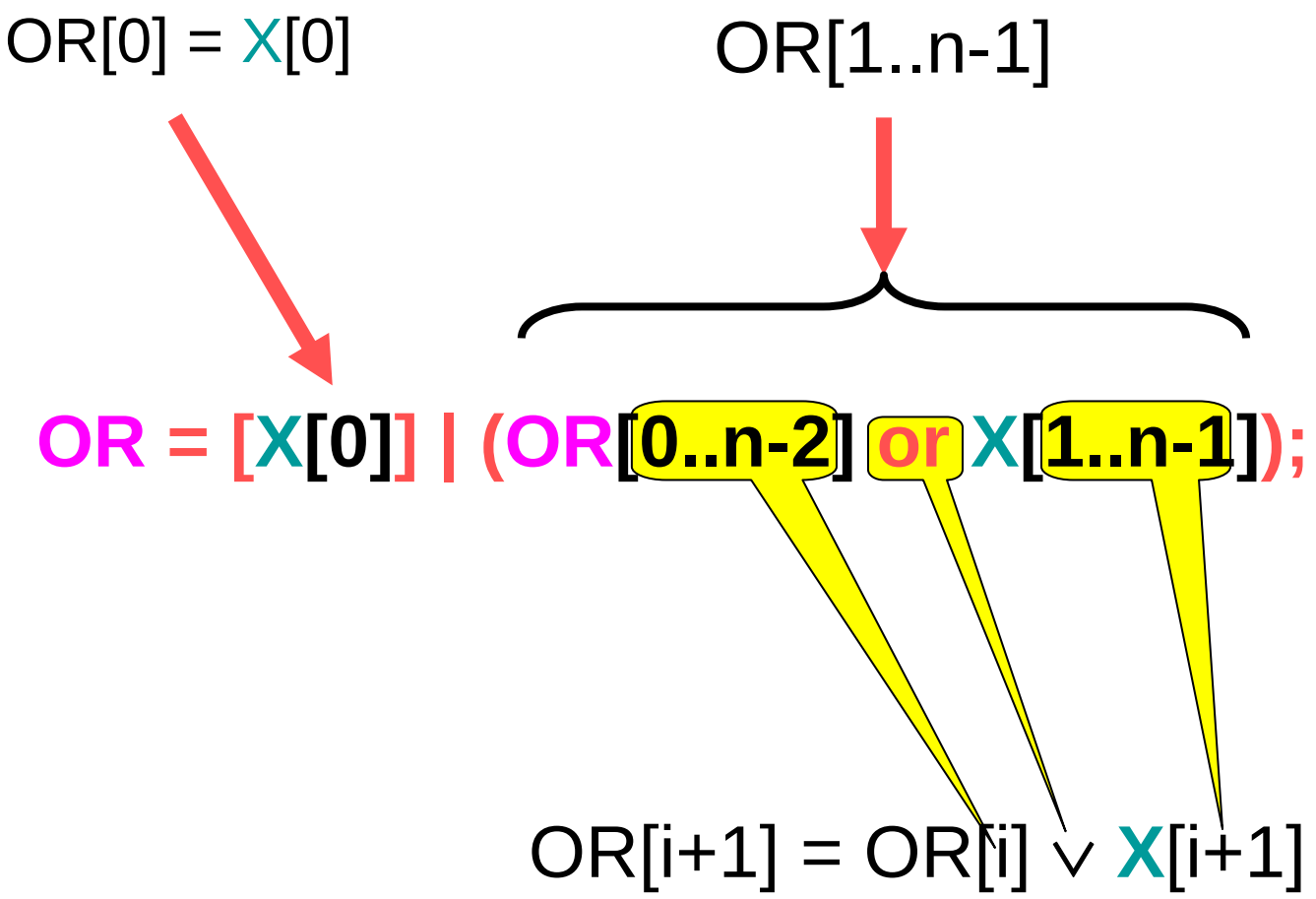
$$OR[i] = \bigvee_{j \le i} X[j]$$

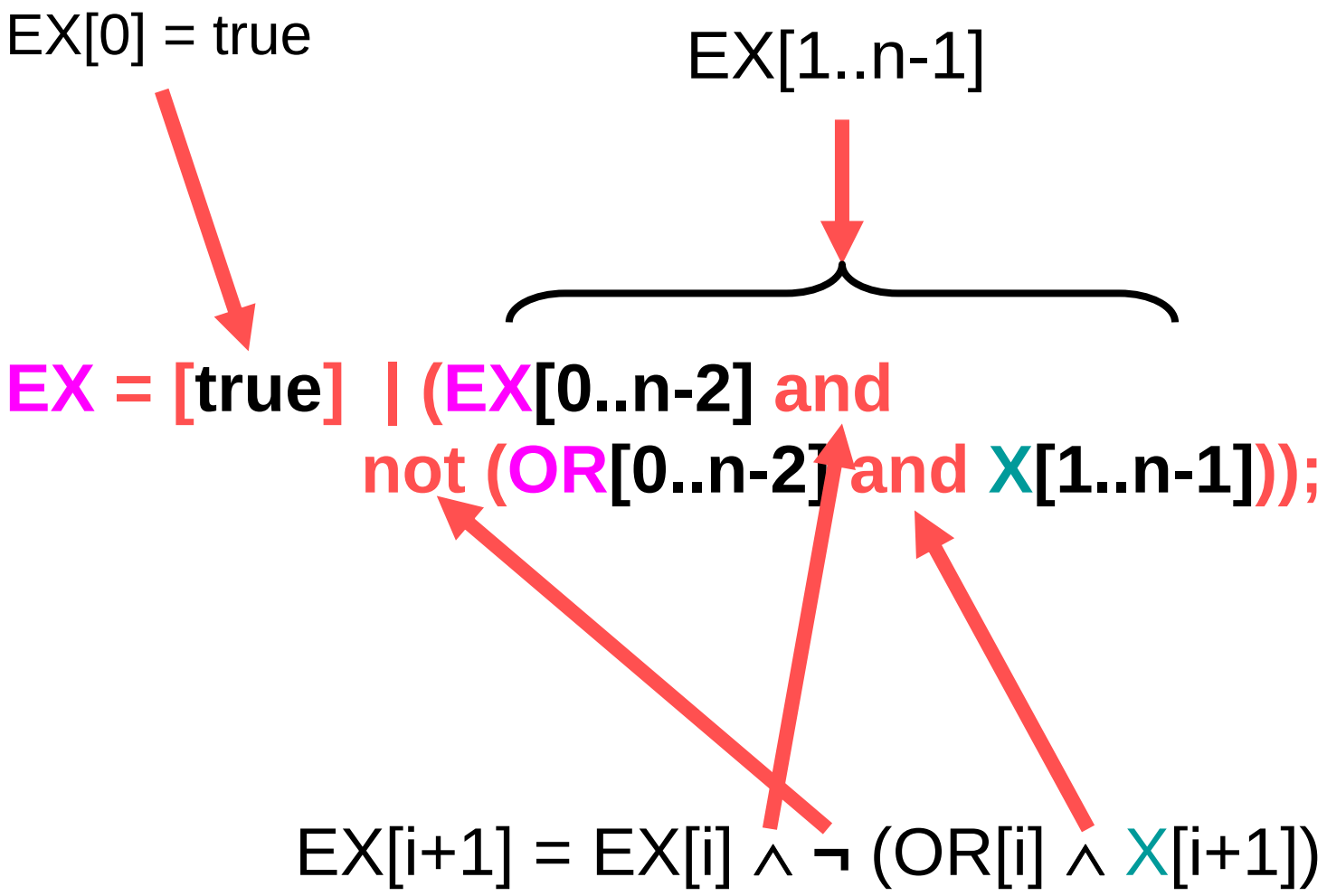
Au moins 1 à true

En terme d'équations de récurrence :

EX[i+1] = EX[i] $\wedge$ $\neg$ (OR[i] $\wedge$ X[i+1]) avec EX[0] = true

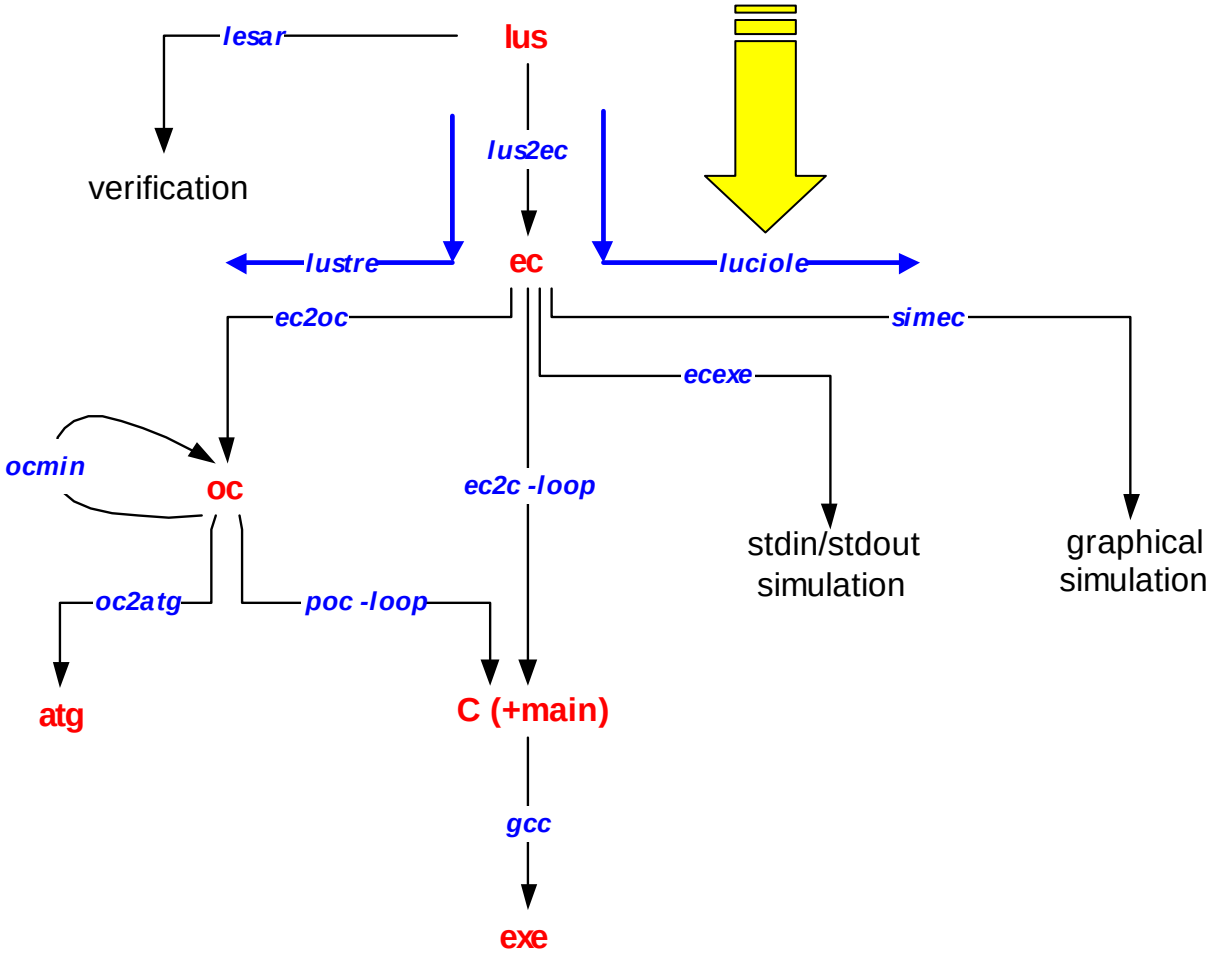
OR[i+1] = OR[i] $\vee$ X[i+1] avec OR[0] = X[0]


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] ∨ 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] ∧ ¬ (OR[i] ∧ 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;
```

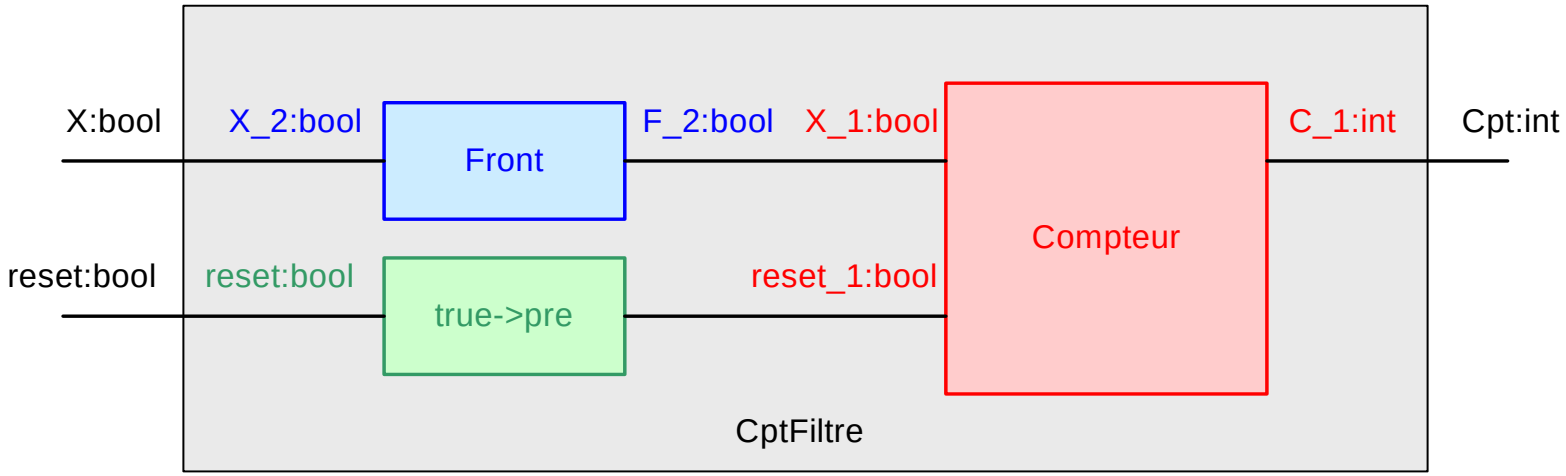
Outils


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 lustre

```
node Compteur(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 Front(X:bool) returns (F:bool);
let
        F = X -> X and not pre X;
tel

node CptFiltre(X, reset:bool) returns (cpt:int);
let
        cpt = Compteur(Front(X), (true -> pre reset));
tel
```

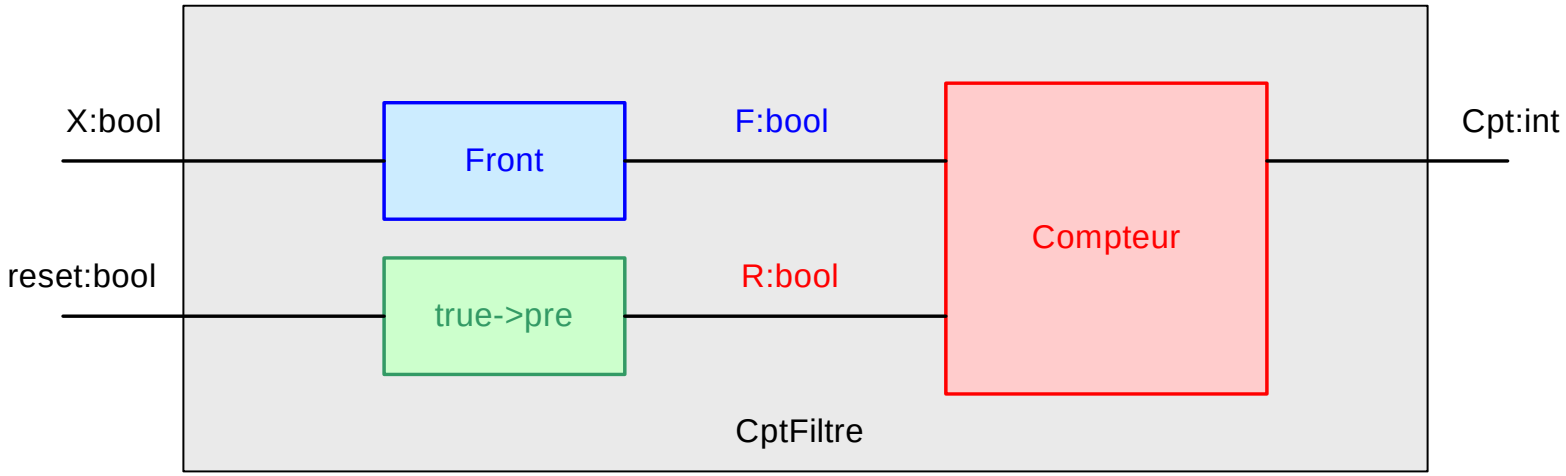
Compilation : expansion (1)



Compilation : expansion (2)

```
node CptFiltre(X, reset:bool) returns (cpt:int);
var
        X_1, reset_1:bool; -- les entrées de Compteur
        C_1:int; -- la sortie de Compteur
        X_2, F_2:bool; -- l'entrée et la sortie de Front
let
        -- appel de Compteur :
        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;
        -- appel de Front :
        F_2 = X_2 -> X_2 and not pre X_2;
        X_2 = X;
tel
```

Compilation : expansion (3)



Compilation : expansion(4)

```
node CptFiltre(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 : vers impératif

avec

pcpt pour **"pre cpt"**

pX pour **"pre X"**

preset pour **"pre reset"**

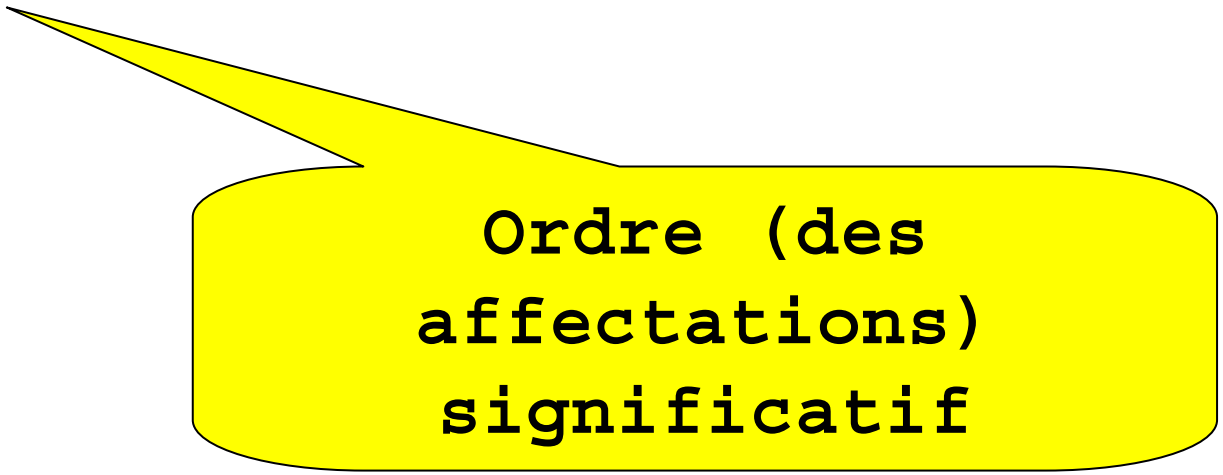
mémoires

```
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);
```

affectation

Compilation : séquentialisation

```
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);
```



Compilation : C code (1)

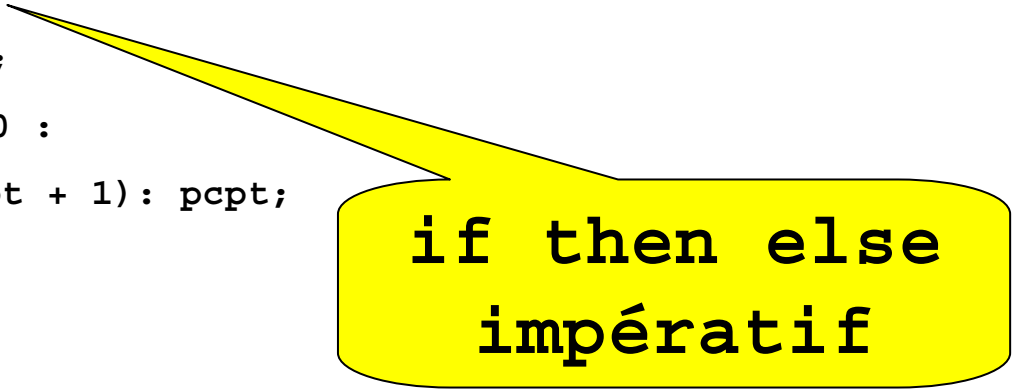
```
/* Les mémoires sont déclarées globalement car
leurs valeurs doivent être rémanentes d'un cycle sur l'autre */
static int pcpt;
static bool pX;
static bool preset;
static bool init = true; /* Initialisation */

/* La procédure qui implémente un cycle de calcul */
void CptFiltre_step (
        bool X; bool reset; /* les entrées, passées par valeur */
        int* cpt /* la sortie, passée par référence */
        )
```

Compilation : C code (2)

```
{
        bool R, F; /* Les variables locales, non rémanentes */
        /* Calcul des variables locales et de la sortie */
        R = init? true : preset;
        F = init? X : (X && ! pX);
        *cpt = R? 0 :
                F? ((init? 0 : pcpt) + 1) :
                        init? 0 : pcpt;
        /* Mémorisations */
        pcpt = *cpt;
        pX = X;
        preset = reset;
        init = false;
}
```

Compilation : C code (3)

```
{
        bool F; /* variable locale, non rémanente */
        if (init) {
                /* Code simplifié pour init = true; F inutile */
                *cpt = 0;
                /* Cette mémorisation n'est nécessaire qu'une fois */
                init = false;
        } else {
                F = (X && ! pX);
                *cpt = preset? 0 :
                        F? (pcpt + 1): pcpt;
        }
        /* Mémorisations */
        pcpt = *cpt;
        pX = X;
        preset = reset;
}
```



Compilation (format)

Boucle simple

- Taille du code linéaire par rapport à la taille du code source
- Exécution systématique du corps de boucle complet : peut être lent

Automate

- Taille du code peut être exponentielle
- Exécution dépendant de l'état courant (switch) : rapide

SIGNAL

Signal

- Un **signal** est une séquence de valeurs associée à une **horloge**.
- Domaines de données:
  - types scalaires (**`boolean, integer, float`**)
  - tableaux de dim arbitraire, avec éléments scalaires
  - type **`event`** qui ne peut être que présent ou absent
- Une horloge est un ensemble discret d ’instants pris dans un ensemble totalement ordonné.

Soit **X** un signal on associe une horloge $\mathbf{C_X}$ qui définit la séquence des instants où **X** est présent.

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

Pour une horloge **C** on définit

$$0_C = \min\{ t \mid t \in C \}$$

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

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

Opérateurs

Les signaux sont définis par des processus élémentaires écrits à l'aide de deux sortes d'opérateurs :

- les opérateurs usuels étendus aux séquences

Y := f(X1, …, Xn) Y,X1, …, Xn même horloge

- des opérateurs temporels :
  - le retard Y := X $ k

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

  - l'extraction Y:=X when B (B signal booléen)

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

- la fusion déterministe

Y := X default Z $\quad C_Y = C_X \cup C_Z$

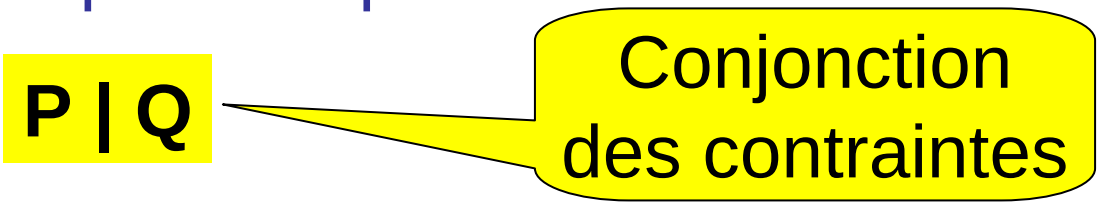
$$\forall t \in C_X \cup C_Z,\ Y_t = \begin{cases} X_t & \text{if } t \in C_X \\ Z_t & \text{autrement} \end{cases}$$

Conséquence : les horloges en Signal ne sont pas de simples arbres, comme en Lustre.

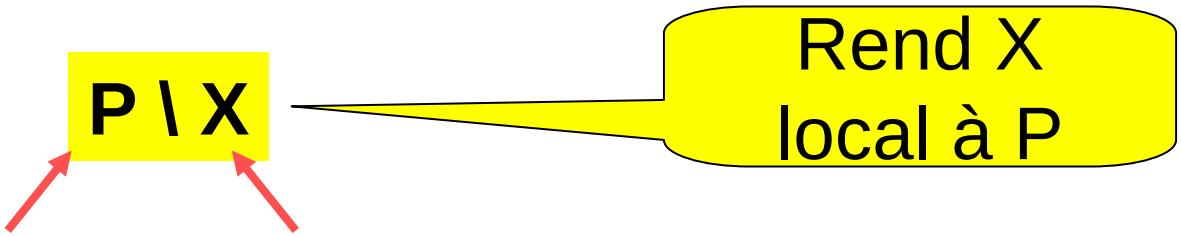
Composition

Les processus se composent à l'aide de deux opérateurs :

- la composition parallèle



- la restriction



+ opérateur synchro

Exemple de prog. Signal

Le signal **MIN** doit être émis toutes les 60 **SEC**

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

Calcul des Horloges

- Les relations entre horloges sont encodées sous forme de polynômes dans le corps $\mathbb{Z}/3\mathbb{Z}$
- Absent $\leftrightarrow$ 0
- Booléen présent **`true`** $\leftrightarrow$ 1
- Booléen présent **`false`** $\leftrightarrow$ -1

Horloges (suite)

Pour un flot booléen s, $-s-s^2$ = 1 ssi s est présent et **`true`**.

`v when s` $\Rightarrow$ produit du polynôme de v par $-s-s^2$

Le polynôme $1-s^2$ vaut 0 ssi s est présent.

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

Pour les flots non booléens, toutes les valeurs présentes représentées par 1, les absentes par 0.