Langage Synchrone Lustre

Nil Geisweiller

Office National d'Étude et de Recherche Aérospatiales

DESS CAMSI 2004

Objectif du cours

Objectif du cours

1. Exposer les principes de la programation des langages synchrones pour le temps réel à travers l'étude approfondi du langage Lustre.

Objectif du cours

1. Exposer les principes de la programation des langages synchrones pour le temps réel à travers l'étude approfondi du langage Lustre.
2. Présenter de manière abordable et pratique la théorie de la preuve de programme à travers l'outil Lesar, un prouveur pour Lustre.

Objectif du cours

1. Exposer les principes de la programation des langages synchrones pour le temps réel à travers l'étude approfondi du langage Lustre.
2. Présenter de manière abordable et pratique la théorie de la preuve de programme à travers l'outil Lesar, un prouveur pour Lustre.
3. Mise en pratique : 3 séances de TPs sur Lustre, avec embarquement du code à bord d'un robot lego.

Table des Matières

Table des Matières

Table des Matières

Table des Matières

Table des Matières

Table des Matières

D'où vient-il

La conception de Lustre a commencé en 1984 dans le laboratoir de recherche VERIMAG à Grenoble. Il est nait d'une collaboration entre informatiens et automaticiens qui voulaient simplifier et même automatiser l'implémentation de lois de commandes.

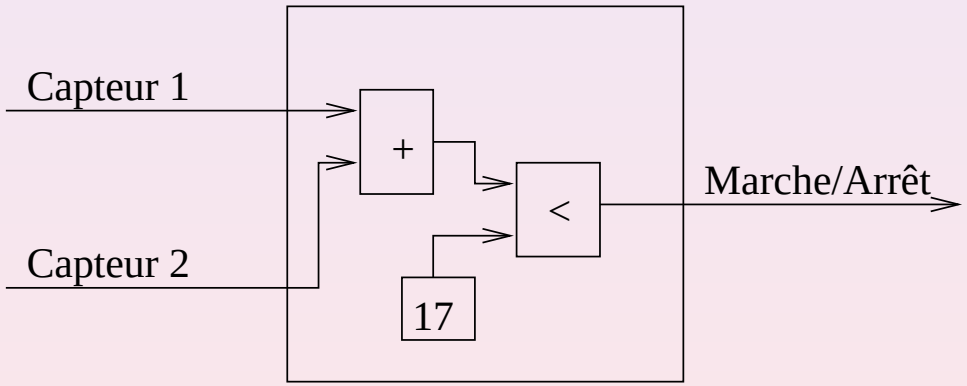


FIG.: *Le schémas de commande d'un thermosta*

Que fait-il ?

Un programme Lustre définie un traitement sur des *flots de données*, c'est à dire qu'il définie quelles sont les valeurs de sorties en fonction des valeurs entrées à tout instant. Il est parfaitement adapté pour programmer des tâches *temps réel* et il est basé sur la notion de *synchronisme fort*. Ce qui fait la force de Lustre se résume en deux points :

1. Simplicité et clarté du langage, approche déclarative.
2. Il est accompagné de puissants outils de vérification formelle grâce à sa sémantique mathématique simple et claire.

Qui l'utilise ?

Il est utilisé de plus en plus dans l'industrie pour des applications qui doivent travailler en temps réel et garantir une sécurité maximale. Airbus dans les A380, Schneider Electric dans les centrals nucléaires.

Table des Matières

Lustre traite des flots de données

Lustre travaille sur des flots de données. S'oppose à Esterel (un autre langage synchrone) qui travaille sur des évenements.

Exemple

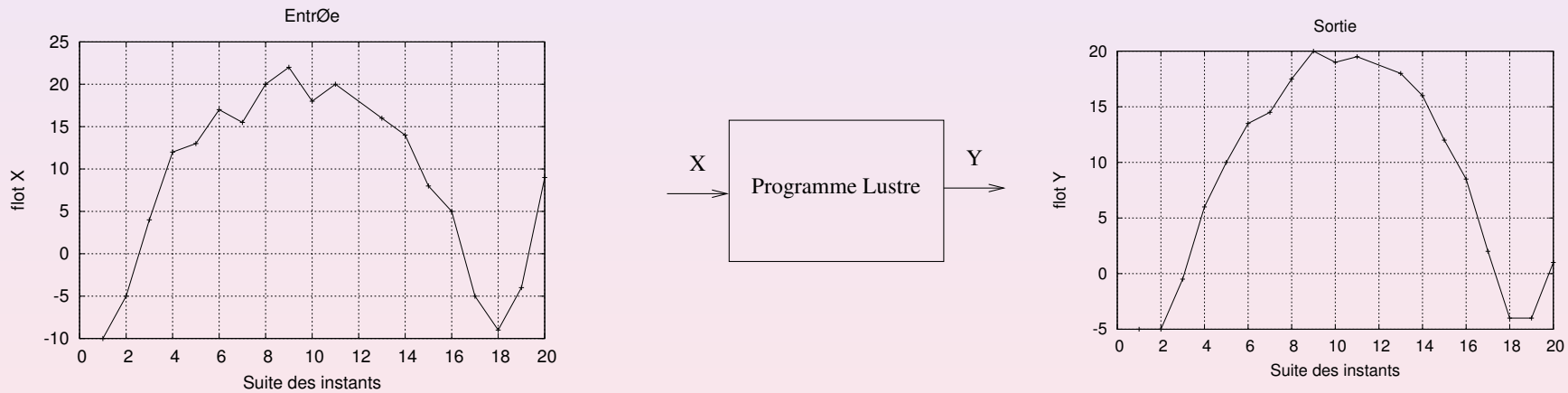


FIG.:

Traitement de flots de données, par exemple un filtre

Le synchronisme fort

Le synchronisme fort

- La notion de *durée* en Lustre *n'existe pas*, seule la notion d'*instant*.

Le synchronisme fort

- La notion de *durée* en Lustre *n'existe pas*, seule la notion d'*instant*.
- Le *temps* en Lustre se résume en une suite infinie d'instants.

Le synchronisme fort

- La notion de *durée* en Lustre *n'existe pas*, seule la notion d'*instant*.
- Le *temps* en Lustre se résume en une suite infinie d'instants.

Le synchronisme fort

- La notion de *durée* en Lustre *n'existe pas*, seule la notion d'*instant*.
- Le *temps* en Lustre se résume en une suite infinie d'instants.

Pour comprendre et interpréter la signification d'un programme il n'y aura jamais besoin d'introduire la notion de durée. Dès lors on pourra considérer sans contradictions internes que tous les temps de calculs sont nulles (c'est en fait ce que considère le model mathématique sous-jacent). Il ne s'agit bien entendu que d'une considération dans la mesure où dans la réalité les calcules on une certaines durées.

Le synchronisme fort

- La notion de *durée* en Lustre *n'existe pas*, seule la notion d'*instant*.
- Le *temps* en Lustre se résume en une suite infinie d'instants.

Pour comprendre et interpréter la signification d'un programme il n'y aura jamais besoin d'introduire la notion de durée. Dès lors on pourra considérer sans contradictions internes que tous les temps de calculs sont nulles (c'est en fait ce que considère le model mathématique sous-jacent). Il ne s'agit bien entendu que d'une considération dans la mesure où dans la réalité les calcules on une certaines durées.
L'hypothèse du temps de calcul nul s'appelle le *synchronisme fort*.

Approche déclarative

Lustre suit une approche *déclarative*, c'est à dire qu'un programme Lustre, contrairement à un programme C par exemple, consiste en un *ensemble de déclarations définissant les relations entre flots d'entrées et flots de sorties*, en d'autres termes des *équations mathématiques*.

Approche déclarative

Lustre suit une approche *déclarative*, c'est à dire qu'un programme Lustre, contrairement à un programme C par exemple, consiste en un *ensemble de déclarations définissant les relations entre flots d'entrées et flots de sorties*, en d'autres termes des *équations mathématiques*.

Exemple

Soit $X = (x_1, x_2, \ldots)$, $Y = (y_1, y_2, \ldots)$ *deux flots de réels d'entrées et* $Z = (z_1, z_2, \ldots)$ *un flots de réel de sorties, alors la déclaration :* $Z = X + Y$ *signifie pour tout instant* $n \in \mathbb{N}^*$ $z_n = x_n + y_n$.

Approche déclarative

Lustre suit une approche *déclarative*, c'est à dire qu'un programme Lustre, contrairement à un programme C par exemple, consiste en un *ensemble de déclarations définissant les relations entre flots d'entrées et flots de sorties*, en d'autres termes des *équations mathématiques*.

Exemple

Soit $X = (x_1, x_2, \ldots)$, $Y = (y_1, y_2, \ldots)$ *deux flots de réels d'entrées et* $Z = (z_1, z_2, \ldots)$ *un flots de réel de sorties, alors la déclaration :* $Z = X + Y$ *signifie pour tout instant* $n \in \mathbb{N}^*$ $z_n = x_n + y_n$.

Alors qu'en langage C le symbole $=$ représente l'affectation, ici il représente l'*égalité mathématique*, écrire $X = X + 1$ sera donc rejeté par le compilateur Lustre.

Determinisme

Un programme Lustre est *déterministe*[1], en ce sens que les flots de sorties sont totalement déterminée par les flots d'entrées.

[1] sauf cas particulier du à des incertitudes pendant l'initialisation

Determinisme

Un programme Lustre est *déterministe*[1], en ce sens que les flots de sorties sont totalement déterminée par les flots d'entrées.
Ce déterminisme est du à une volonté des concepteurs de Lustre d'offrire un langage où les résultats attendus sont uniques et bien définies.Techniquement, cela vient :

[1] sauf cas particulier du à des incertitudes pendant l'initialisation

Determinisme

Un programme Lustre est *déterministe*[1], en ce sens que les flots de sorties sont totalement déterminée par les flots d'entrées.
Ce déterminisme est du à une volonté des concepteurs de Lustre d'offrire un langage où les résultats attendus sont uniques et bien définies.Techniquement, cela vient :

1. d'une part, d'une certaine restriction dans l'écriture des équations décrivant les flots de données du programme.

[1] sauf cas particulier du à des incertitudes pendant l'initialisation

Determinisme

Un programme Lustre est *déterministe*[1], en ce sens que les flots de sorties sont totalement déterminée par les flots d'entrées.
Ce déterminisme est du à une volonté des concepteurs de Lustre d'offrire un langage où les résultats attendus sont uniques et bien définies.Techniquement, cela vient :

1. d'une part, d'une certaine restriction dans l'écriture des équations décrivant les flots de données du programme.
2. et d'autres part d'une conséquence du synchronisme fort de Lustre.

[1] sauf cas particulier du à des incertitudes pendant l'initialisation

Table des Matières

Avant de rentrer dans le détail

- Un programme Lustre définit un réseau de composants travaillant sur des flots. En Lustre il est possible de définir ses propres composants, on les appelle *nœuds*.

Avant de rentrer dans le détail

- Un programme Lustre définit un réseau de composants travaillant sur des flots. En Lustre il est possible de définir ses propres composants, on les appelle *nœuds*.
- Un programme Lustre consiste en un nœud principale et éventuellement d'autres nœuds auxilières.

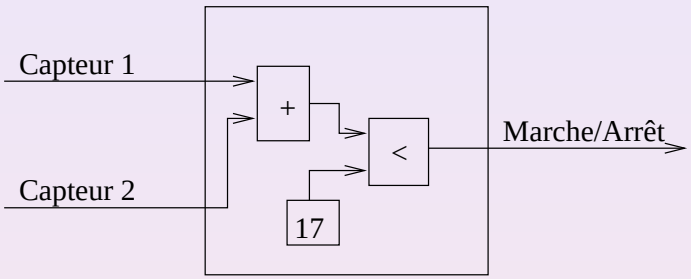
Avant de rentrer dans le détail

- Un programme Lustre définit un réseau de composants travaillant sur des flots. En Lustre il est possible de définir ses propres composants, on les appelle *nœuds*.
- Un programme Lustre consiste en un nœud principale et éventuellement d'autres nœuds auxilières.
- L'entête d'un nœud contient la déclaration des flots d'entrées, des flots de sorties, des flots internes. Le corps d'un nœud contient les équations définissant les flots de sorties et internes en fonction des flots d'entrées.

Structure d'un programme Lustre

```
[declaration de types et de fonctions externes]
node nom(déclaration des flots d'entrées)
   returns(déclaration des flots de sorties) ;
[var declaration des flots internes]
let
[assertions]
  système d'équations définissant une
  et une seule fois les flots internes et de sorties
  en fonction d'eux mêmes et des flots d'entrées
tel
[autres nœuds]
```

Un exemple de programme Lustre

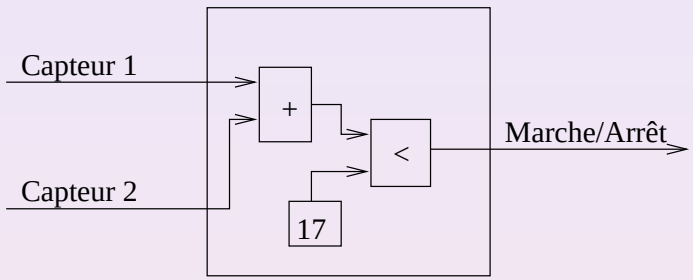



```
node thermos(C1,C2 : real)
    returns(M_A : bool) ;
let
  M_A=((C1+C2)<17) ;
tel
```

FIG.:

schémas et code Lustre d'un thermosta

Un exemple de programme Lustre




```
node thermos(C1,C2 : real)
    returns(M_A : bool) ;
let
  M_A=((C1+C2)<17) ;
tel
```

FIG.:

schémas et code Lustre d'un thermosta

Exemple d'une exécution :

temps	1	2	3	4	...
C1	10	13	15	12.6	...
C2	5	5.5	4	3	...
M_A	vrai	faux	faux	vrai	...

Les types de données

Les types de données transportés par les flots sont :

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : int, bool, real

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : int, bool, real
- Les tableaux : int^ 3, real^ 5^ 2, [int, bool, [int, real]], ...

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : int, bool, real
- Les tableaux : int^ 3, real^ 5^ 2, [int, bool, [int, real]], ...
- Les n-uplets (bool,bool), (int, bool, real), ... *(type particulier qui n'a pas besoin d'être déclaré, il est manipulé directement dans le système d'équation)*

Les types de données

Les types de données transportés par les flots sont :

- Les types de bases : int, bool, real
- Les tableaux : int^ 3, real^ 5^ 2, [int, bool, [int, real]], ...
- Les n-uplets (bool,bool), (int, bool, real), ... *(type particulier qui n'a pas besoin d'être déclaré, il est manipulé directement dans le système d'équation)*

Et c'est tout ! Construction de type assez limité, pas de récursivité, mais il reste la possibilité d'importer des types de l'extérieur.

Dans le but de tenir l'hypothèse de synchronisme fort le langage ne contient aucune structure qui pourrait rendre non borné le temps de calcul, comme par exemple les fonctions récursives ou les types récurcifs, les boucles, etc.

Un système d'équations en mathématiques aboutit en général à trois possibilités :

Un système d'équations en mathématiques aboutit en général à trois possibilités :

1. le système comporte plusieurs solutions (par exemple, pas assez d'équations pour trop d'inconnues)

Un système d'équations en mathématiques aboutit en général à trois possibilités :

1. le système comporte plusieurs solutions (par exemple, pas assez d'équations pour trop d'inconnues)
2. le système ne comporte aucune solution (par exemple, informations contradictoires au sein du système)

$$\begin{cases} \dots \\ X = 3 \\ X = 4 \\ \dots \end{cases}$$

Un système d'équations en mathématiques aboutit en général à trois possibilités :

1. le système comporte plusieurs solutions (par exemple, pas assez d'équations pour trop d'inconnues)
2. le système ne comporte aucune solution (par exemple, informations contradictoires au sein du système)

$$\begin{cases} \dots \\ X = 3 \\ X = 4 \\ \dots \end{cases}$$

3. le système comporte une et une seule solution pour chaque variable (par exemple, système linéaire dont le déterminant est non nul)

Afin de forcer l'utilisateur à écrire uniquement des systèmes déterministes (c'est-à-dire avec une et une seule solution par flot de données interne ou de sortie), Lustre n'accepte dans sa syntaxe que les systèmes d'équations de la forme :

$$\begin{cases} \ldots & \\ Y & = \ldots \\ Z & = \ldots \\ \ldots & \end{cases}$$

où Y, Z, ... sont les flots internes ou de sorties, définis chacun une est une seule fois et de manière non circulaire.

Afin de forcer l'utilisateur à écrire uniquement des systèmes déterministes (c'est-à-dire avec une et une seule solution par flot de données interne ou de sortie), Lustre n'accepte dans sa syntaxe que les systèmes d'équations de la forme :

$$\begin{cases} \ldots & & \\ Y & = & \ldots \\ Z & = & \ldots \\ \ldots & & \end{cases}$$

où Y, Z, ... sont les flots internes ou de sorties, définis chacun une est une seule fois et de manière non circulaire.

Interdit $\begin{cases} Y & = & Z \\ Z & = & Y \end{cases}$

Les opérateurs classiques

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs logiques :

- Binaire : `or, xor, and, =>`
- Unaire : `not`

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs logiques :

- Binaire : `or, xor, and, =>`
- Unaire : `not`

Les opérateurs de comparaison : =, <>, <, >, <=, >=

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : $+$, -, *, div, mod, /
- Unaire : -

Les opérateurs logiques :

- Binaire : `or, xor, and, =>`
- Unaire : `not`

Les opérateurs de comparaison : $=$, $<>$, $<$, $>$, $<=$, $>=$
L'opérateurs de contrôle `if . then . else .`, peut se traduire *"quand . alors . sinon ."* au lieu de *si . alors . sinon .*

Les opérateurs classiques

Les opérateurs arithmétiques :

- Binaire : +, -, *, div, mod, /
- Unaire : -

Les opérateurs logiques :

- Binaire : `or, xor, and, =>`
- Unaire : `not`

Les opérateurs de comparaison : =, <>, <, >, <=, >=
L'opérateurs de contrôle `if . then . else .`, peut se traduire *"quand . alors . sinon ."* au lieu de *si . alors . sinon .*
Les constantes :

- true = (true, true, ...)
- 0 = (0, 0, ...)
- 1.45 = (1.45,1.45, ...)

Exemple : un nœud qui additionne ou soustrait deux flots d'entiers :

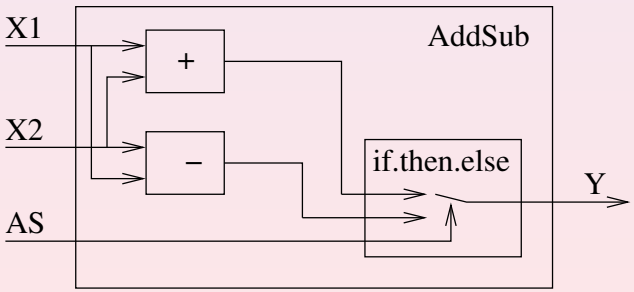
Soit AS un flot d'entrée booléen,
Soit X1, X2 deux flots d'entrées entiers,
Soit Y un flot de sortie entier,

$$Y = \begin{cases} X1 + X2 & \text{quand AS est vrai} \\ X2 - X1 & \text{autrement} \end{cases}$$

Exemple : un nœud qui additionne ou soustrait deux flots d'entiers :

Soit AS un flot d'entrée booléen,
Soit X1, X2 deux flots d'entrées entiers,
Soit Y un flot de sortie entier,

$$Y = \begin{cases} X1 + X2 & \text{quand AS est vrai} \\ X2 - X1 & \text{autrement} \end{cases}$$




```
node AddSub(AS :bool ;X1,X2 :int)
    returns(Y :int) ;
let
  Y = if AS then X1+X2
              else X2-X1 ;
tel
```

Exercice

Ecrire un programme Lustre qui prend en entrée un flot réel E, en sortie deux flots : un flots booléen B qui devient vrai quand l'entrée dépasse 100 et un flot de sortie réel S qui renvoie 0 quand B est faux et qui renvoie la différence E-100 quand B est vrai.

Le prototype du nœud est le suivant :

```
node noeud(E : real)returns(B :bool ;S :real) ;
```

Les flots de données et les horloges

Les flots de données et les horloges

Définition

Un flot de données est un couple :

1. *d'une suite infinie de valeurs d'un certain type T, noté $(x_1, x_2, \ldots, x_n, \ldots)$*
2. *d'une horloge.*

Les flots de données et les horloges

Définition

Un flot de données est un couple :

1. *d'une suite infinie de valeurs d'un certain type T, noté $(x_1, x_2, \ldots, x_n, \ldots)$*
2. *d'une horloge.*

Définition

Une horloge est ou bien :

- *un flot de données de type booléennes.*
- *une horloge de base.*

Les flots de données et les horloges

Définition

Un flot de données est un couple :

1. *d'une suite infinie de valeurs d'un certain type T, noté $(x_1, x_2, \ldots, x_n, \ldots)$*
2. *d'une horloge.*

Définition

Une horloge est ou bien :

- *un flot de données de type booléennes.*
- *une horloge de base.*

Notez bien qu'il s'agit d'une définition mutuellement récursive de flot est de horloge.

Les flots de données et les horloges

Définition

Les flots qui ont pour horloge l'horloge de base sont dit continus, les autres sont dit semi-continus.

Les flots de données et les horloges

Définition

Les flots qui ont pour horloge l'horloge de base sont dit continus, les autres sont dit semi-continus.

Chaque position de la suite de valeurs du flot correspond à la notion intuitive d'*instant*.

Les flots de données et les horloges

Définition

Les flots qui ont pour horloge l'horloge de base sont dit continus, les autres sont dit semi-continus.

Chaque position de la suite de valeurs du flot correspond à la notion intuitive d'*instant*.
La sémantique intuitive d'une horloge sera la suivante : un flot est définie aux instants où son horloge est vraie. Pour l'horloge de base le flot est toujour définie. Il est important de remarquer qu'étant donnée qu'une horloge est un flot, celle-ci possède également une horloge et ainsi de suite jusqu'à l'horloge de base.

Exemples

- Un flot d'entiers sur l'horloge de base : $(4, 2, \ldots, 7, \ldots)$

Exemples

- Un flot d'entiers sur l'horloge de base :
  $(4, 2, \ldots, 7, \ldots)$
- Un flot de booléens, noté B, sur l'horloge de base :
  $B = (\mathit{vrai}, \mathit{faux}, \mathit{vrai}, \mathit{faux}, \ldots)$

Exemples

- Un flot d'entiers sur l'horloge de base :
  $(4, 2, \ldots, 7, \ldots)$
- Un flot de booléens, noté B, sur l'horloge de base :
  $B = (vrai, faux, vrai, faux, \ldots)$
- Un flot de réels sur l'horloge B, B est un flot booléen c'est donc bien une horloge :
  1. $(23.4, 25, 01, 12.3, 1.2, \ldots)$ pour la suite infinie de réels,
  2. B pour l'horloge,

  ce qui peut aussi se noter : $(23.4, \_, 12.3, \_, \ldots)$

Les opérateurs temporels

Les opérateurs temporels

- pre (précédent) : opérateur permettant de travailler sur le passé d'un flot

Les opérateurs temporels

- pre (précédent) : opérateur permettant de travailler sur le passé d'un flot
- $->$ (suivi de) : opérateur permettant d'initialiser un flot

Les opérateurs temporels

- pre (précédent) : opérateur permettant de travailler sur le passé d'un flot
- $->$ (suivi de) : opérateur permettant d'initialiser un flot
- when (sous-échantillonnage) : opérateur permettant de transformer un flot continu en un flot semi-continu

Les opérateurs temporels

- pre (précédent) : opérateur permettant de travailler sur le passé d'un flot
- $->$ (suivi de) : opérateur permettant d'initialiser un flot
- when (sous-échantillonnage) : opérateur permettant de transformer un flot continu en un flot semi-continu
- current (sur-échantillonnage) : opérateur permettant de transforme un flot semi-continu en un flot continu

Les opérateurs temporels

- pre (précédent) : opérateur permettant de travailler sur le passé d'un flot
- $->$ (suivi de) : opérateur permettant d'initialiser un flot
- when (sous-échantillonnage) : opérateur permettant de transformer un flot continu en un flot semi-continu
- current (sur-échantillonnage) : opérateur permettant de transforme un flot semi-continu en un flot continu

Afin de définir l'horloge d'un flot, on ajoute une nouvelle structure de type : T when B

- T est un type classique
- B est une variable de flot de type *bool* ou de type *bool when B*

Pour simplifier on va d'abord définir nos opérateurs temporels travaillant sur des flots continus, le passage au semi-continu n'étant en faite qu'un changement de point de vue.

L'opérateur pre (précédent)

Il permet de mémoriser la valeur précédente d'un flot ou d'un ensemble de flots.

Soit X le flot $(X_1, \ldots, X_n, \ldots)$ alors :

- $pre(X)$ est le flot $(\omega, X_1, \ldots, X_n, \ldots)$
- $pre(pre(X))$ est le flot $(\omega, \omega, X_1, \ldots, X_n, \ldots)$
- ...

Par extension, l'équation $(Y, Y') = pre(X, X')$ signifie :

- $Y_0 = \omega,\ Y'_0 = \omega$
- pour tout $n > 0,\ Y_n = X_{n-1}$ et $Y'_n = X'_{n-1}$

ω l'indéterminé

ω représente potentiellement n'importe quelle valeur, il représente la seule indétermination qui peut apparaître en Lustre, mais celle-ci ne se manifeste (éventuellement) qu'à l'initialisation du système.

Attention, ω n'est pas un symbol du langage Lustre, c'est seulement une notation de ce cours pour désigner la valeur indéterminée.

L'opérateur $->$ (suivi de)

Il permet d'initialiser un flot ou un ensemble de flots.

Soit X le flot $(X_1, \ldots, X_n, \ldots)$ et Y le flot $(Y_1, Y_2, \ldots, Y_n, \ldots)$ alors :
$X -> Y$ est le flot $(X_1, Y_2, \ldots, Y_n, \ldots)$
Par extension, l'équation $(Z, Z') = (X, X') -> (Y, Y')$ signifie :

- $Z_1 = X_1$, $Z'_1 = X'_1$
- pour tout $n > 0$, $Z_n = Y_n$ et $Z'_n = Y'_n$

Exemple : un nœud qui cumule les données d'entrées

Soit X un flot d'entrées entier
Soit Y un flot de sorties entier

$$(Y_1, Y_2, Y_3, \ldots) = (X_1, X_1 + X_2, X_1 + X_2 + X_3, \ldots)$$

Exemple : un nœud qui cumule les données d'entrées

Soit X un flot d'entrées entier
Soit Y un flot de sorties entier

$$(Y_1, Y_2, Y_3, \ldots) = (X_1, X_1 + X_2, X_1 + X_2 + X_3, \ldots)$$

```
node CumSum(X :int) returns(Y :int);
let
    Y = X -> (X + pre(Y));
tel
```

Exemple : un compteur

Soit *RAS* (remise à zero) un flot d'entrées booléen
Soit *N* un flot de sorties entier

A chaque instant le flot de sortie *N* s'incrémente de 1, quand *RAS* est vrai *N* revient à zero.

Exemple : un compteur

Soit *RAS* (remise à zero) un flot d'entrées booléen
Soit *N* un flot de sorties entier

A chaque instant le flot de sortie *N* s'incrémente de 1, quand *RAS* est vrai *N* revient à zero.

```
node compteur(RAZ :bool) returns(N :int) ;
let
    N = 0 -> if RAZ then 0 else (pre(N)+1) ;
tel
```

Un autre exemple

```
node dec(RAZ :bool) returns(N :int) ;
let
    N = -compteur(RAZ) ;
tel
```

```
node If_Inc_Dec(RAZ,ID :bool) returns(N :int) ;
let
    N = if ID then compteur(RAZ) else dec(RAZ) ;
tel
```

Un autre exemple

```
node dec(RAZ :bool) returns(N :int) ;
let
    N = -compteur(RAZ) ;
tel
```

```
node If_Inc_Dec(RAZ,ID :bool) returns(N :int) ;
let
    N = if ID then compteur(RAZ) else dec(RAZ) ;
tel
```

Exemple

temps	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*. . .*
RAZ	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*faux*	*. . .*
ID	*vrai*	*vrai*	*vrai*	*vrai*	*faux*	*faux*	*faux*	*vrai*	*. . .*
N	*0*	*1*	*2*	*3*	*-4*	*-5*	*-6*	*7*	*. . .*

L'opérateur when : opérateur de sous-échantillonnage

Soit B un flot booléen, soit X un flot de type `T`.

L'équation `Y = X when B ;` définit un flot semi-continu Y de type `T when B` égal à X quand B est vrai indéfinie sinon.

L'opérateur when : opérateur de sous-échantillonnage

Soit B un flot booléen, soit X un flot de type `T`.

L'équation `Y = X when B ;` définit un flot semi-continu Y de type `T when B` égal à X quand B est vrai indéfinie sinon.

Exemple

temps	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	...
X	*2*	*4*	*1*	*8*	*3*	*9*	*3*	*4*	...
B	*vrai*	*faux*	*faux*	*vrai*	*faux*	*vrai*	*faux*	*vrai*	...
Y	*2*	_	_	*8*	_	*9*	_	*4*	...

L'opérateur when : opérateur de sous-échantillonnage

Soit B un flot booléen, soit X un flot de type `T`.

L'équation `Y = X when B ;` définit un flot semi-continu Y de type `T when B` égal à X quand B est vrai indéfinie sinon.

Exemple

temps	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*...*
X	*2*	*4*	*1*	*8*	*3*	*9*	*3*	*4*	*...*
B	*vrai*	*faux*	*faux*	*vrai*	*faux*	*vrai*	*faux*	*vrai*	*...*
Y	*2*	_	_	*8*	_	*9*	_	*4*	*...*

Attention ! Ne pas confondre _ (indéfini) et ω (indeterminé). En effet quand Y est indéfini sa valeur de sortie est inexistante tantdis que quand il est indéterminé sa valeur de sortie est inconnue mais existe.

L'opérateur current : prolongement, sur-échantillonnage

Soit X un flot semi-continu de type `T when B`,
Soit Y un flot de type `T`
L'équation `Y = current X ;` définit un flot Y de type `T` égale à X quand X est définie égale à la dernière valeur définie de X quand X est indéfinie, ou ω s'il n'y a pas de dernière valeur.

L'opérateur current : prolongement, sur-échantillonnage

Soit X un flot semi-continu de type `T when B`,
Soit Y un flot de type `T`
L'équation `Y = current X ;` définit un flot Y de type `T` égale à X quand X est définie égale à la dernière valeur définie de X quand X est indéfinie, ou ω s'il n'y a pas de dernière valeur.

Exemple

temps	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*...*
X	*2*	_	_	*8*	_	*9*	_	*4*	*...*
Y	*2*	*2*	*2*	*8*	*8*	*9*	*9*	*4*	*...*

Exemple : un compteur intermitant

Soit RAZ et B deux flots d'entrées booléens, N un flot de sorties entier. Quand B est vrai le compteur incrémente la sortie N.

Exemple : un compteur intermitant

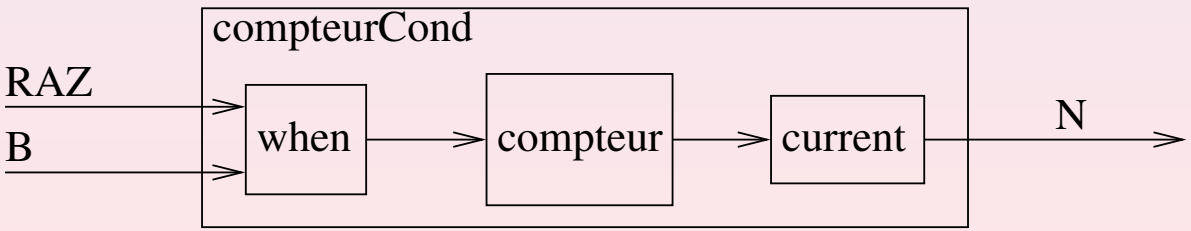
Soit *RAZ* et *B* deux flots d'entrées booléens, *N* un flot de sorties entier. Quand *B* est vrai le compteur incrémente la sortie *N*.

```
node compteurCond(RAZ,B :bool) returns(N :int) ;
let
    N=current(compteur(RAZ when B)) ;
tel
```

Exemple : un compteur intermitant

Soit *RAZ* et *B* deux flots d'entrées booléens, *N* un flot de sorties entier. Quand *B* est vrai le compteur incrémente la sortie *N*.

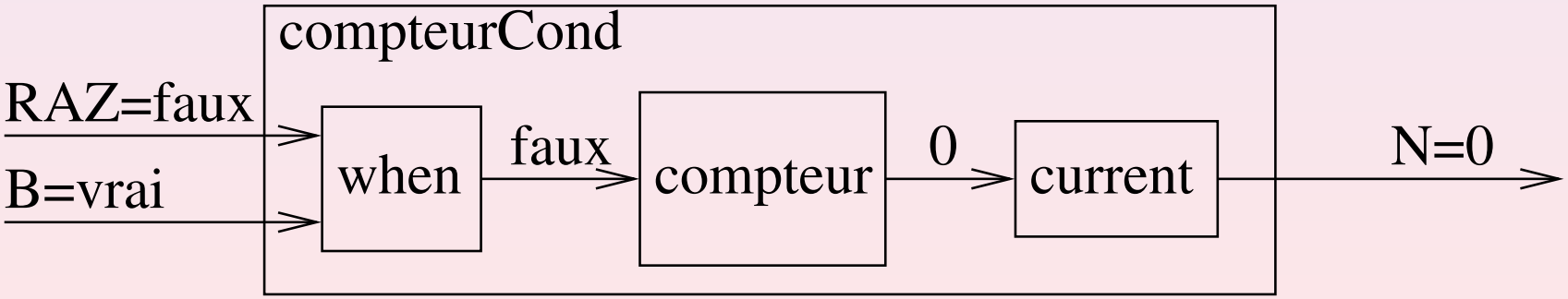
```
node compteurCond(RAZ,B :bool) returns(N :int) ;
let
    N=current(compteur(RAZ when B)) ;
tel
```



Execution de compteurCond :

/

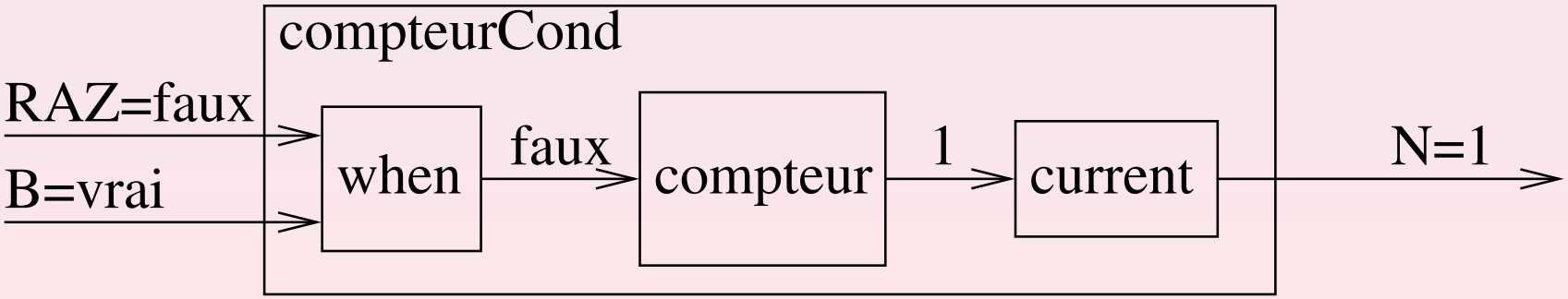
temps=1
RAZ=faux
B=vrai
N=0



Execution de compteurCond :

/-

temps=2
RAZ=faux
B=vrai
N=1



Execution de compteurCond :

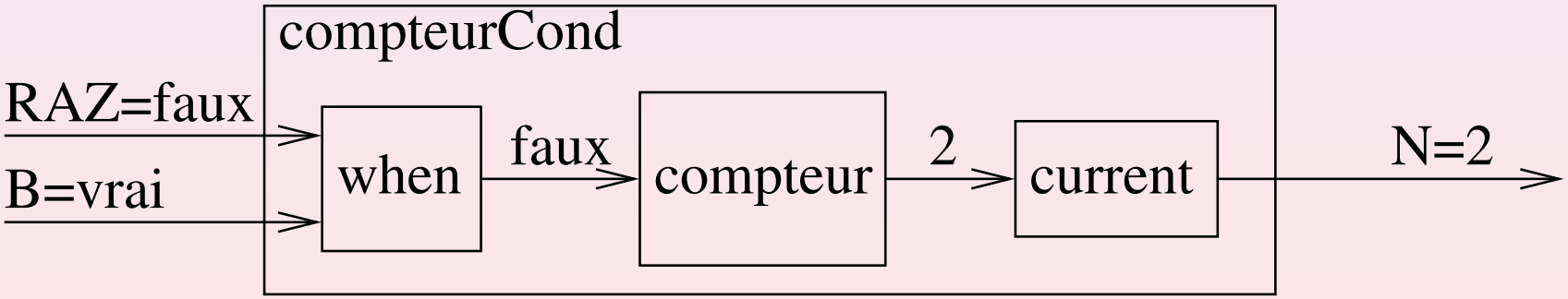
/-\

temps=3
RAZ=faux
B=faux
N=1

Execution de compteurCond :

/-_

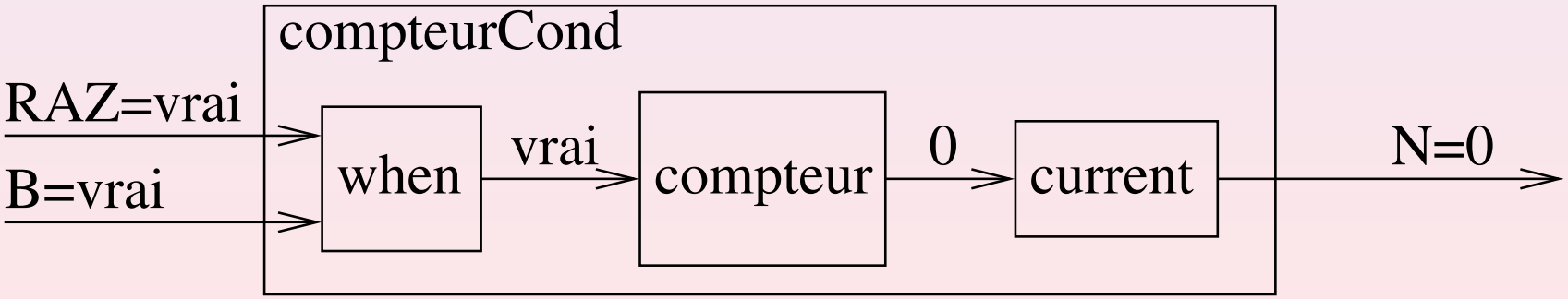
temps=4
RAZ=faux
B=vrai
N=2



Execution de compteurCond :

/-_ /

temps=5
RAZ=vrai
B=vrai
N=0



Execution de compteurCond :

/-_ /-

temps=6
RAZ=faux
B=faux
N=0

Execution de compteurCond :

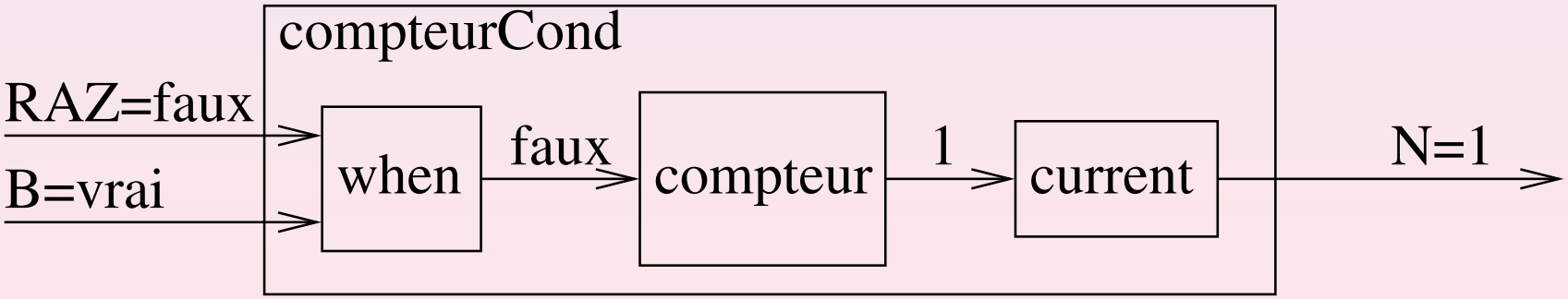
/-_ /-\

temps=7
RAZ=faux
B=faux
N=0

Execution de compteurCond :

/-_ /-_

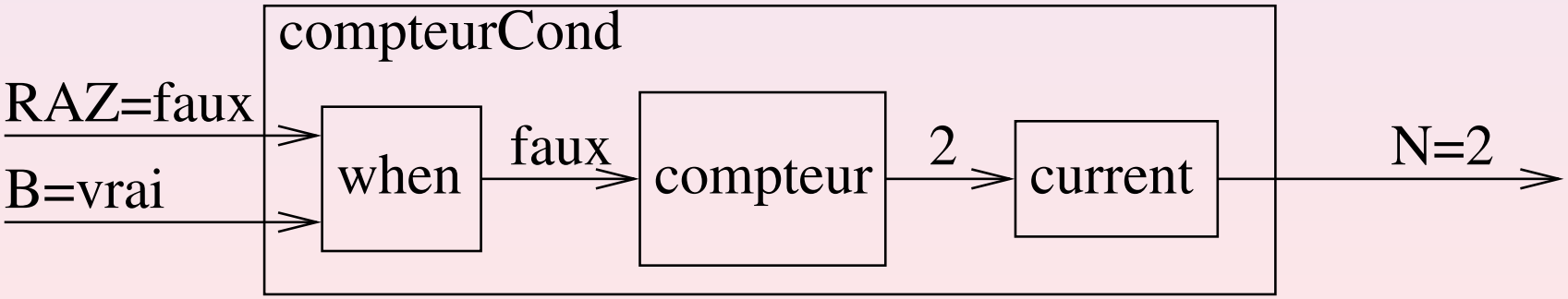
temps=8
RAZ=faux
B=vrai
N=1



Execution de compteurCond :

/-_ /-_ /

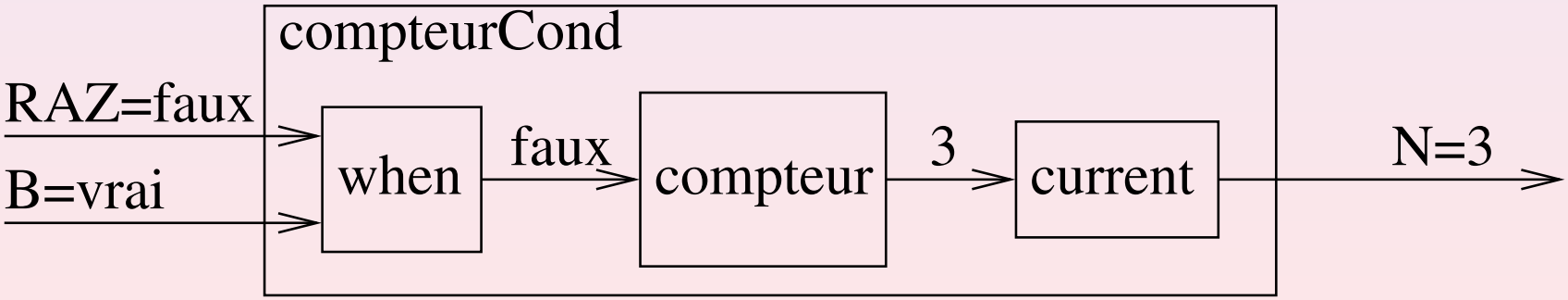
temps=9
RAZ=faux
B=vrai
N=2



Execution de compteurCond :

/-_ /-_ /-

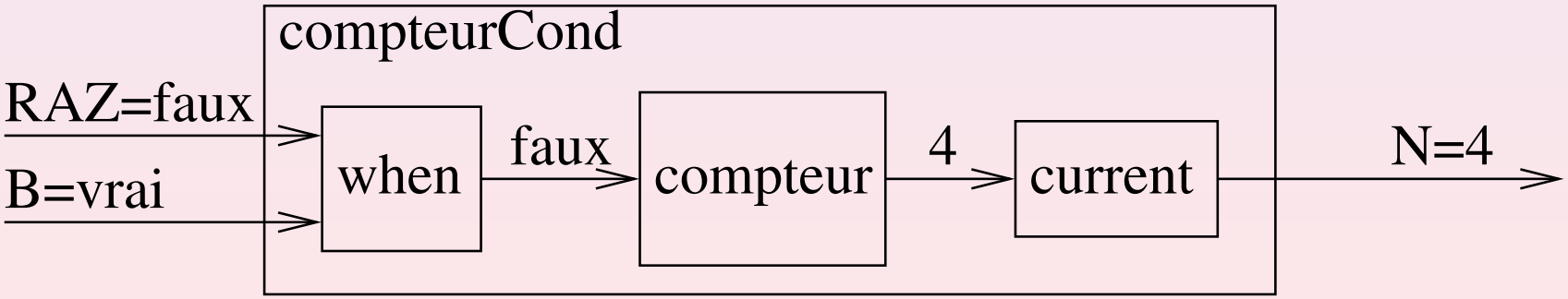
temps=10
RAZ=faux
B=vrai
N=3



Execution de compteurCond :

/-_ /-_ /-\

temps=11
RAZ=faux
B=vrai
N=4



Les opérateurs temporels sur des flots semi-continus

Les opérateurs temporels sur des flots semi-continus

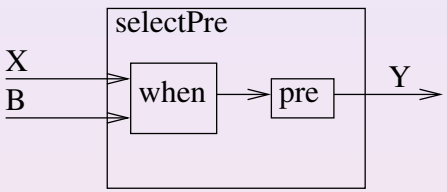
Il faut considérer le principe suivant : le caractère continu ou semi-continu d'un flot n'est une question de point de vue.

Les opérateurs temporels sur des flots semi-continus

Il faut considérer le principe suivant : le caractère continu ou semi-continu d'un flot n'est une question de point de vue.

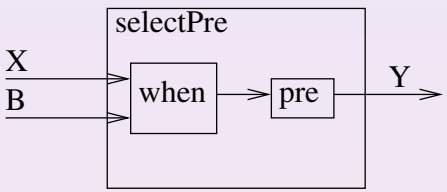
Pour comprendre ce principe on considére l'exemple précédant : du point du nœud `compteurCond` le flot qui rentre dans le nœud `compteur` est semi-continu mais du point de vue du nœud `compteur` ce flot est continu.
De cette manière toutes les définitions des opérateurs temporels pre, $->$, when, current sont transposables aux flots semi-continus si on adopte le bon point vue.

Exemple, l'opérateur pre




```
node selectPre(X :int, B :bool)
     returns(Y :int when B) ;
let
    Y = pre(X when B) ;
tel
```

Exemple, l'opérateur pre




```
node selectPre(X :int, B :bool)
     returns(Y :int when B) ;
let
    Y = pre(X when B) ;
tel
```

Exemple

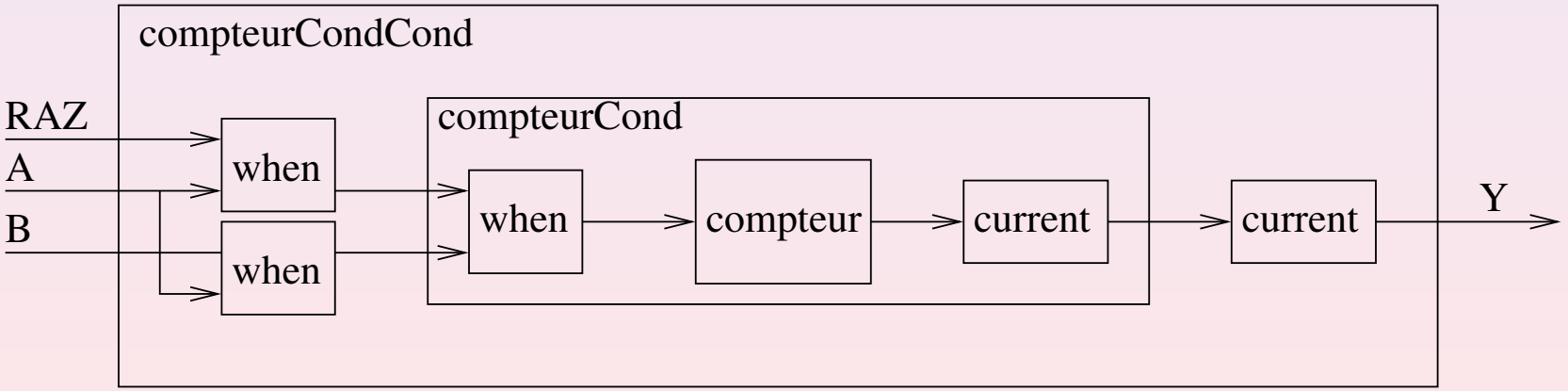
temps	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*. . .*
X	*2*	*1*	*10*	*8*	*5*	*9*	*2*	*4*	*. . .*
B	*vrai*	*faux*	*vrai*	*vrai*	*faux*	*faux*	*faux*	*vrai*	*. . .*
X when B									*. . .*
Y									*. . .*

Exemple, les opérateurs when et current

```
node compteurCondCond(RAZ,A,B :bool) returns(Y :int) ;
let
    Y=current(compteurCond(RAZ when A, B when A)) ;
tel
```

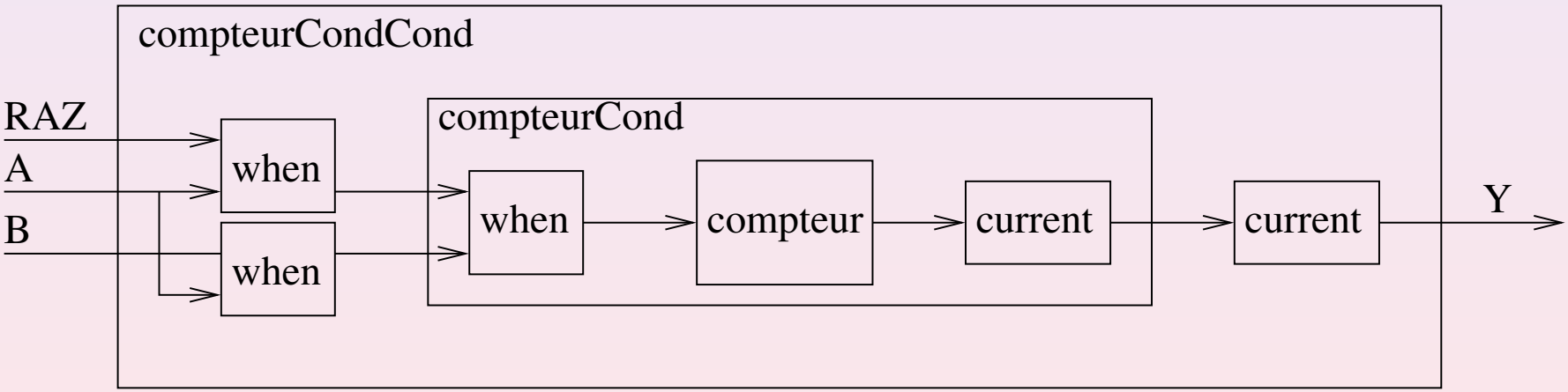
Exemple, les opérateurs when et current

```
node compteurCondCond(RAZ,A,B :bool) returns(Y :int) ;
let
    Y=current(compteurCond(RAZ when A, B when A)) ;
tel
```



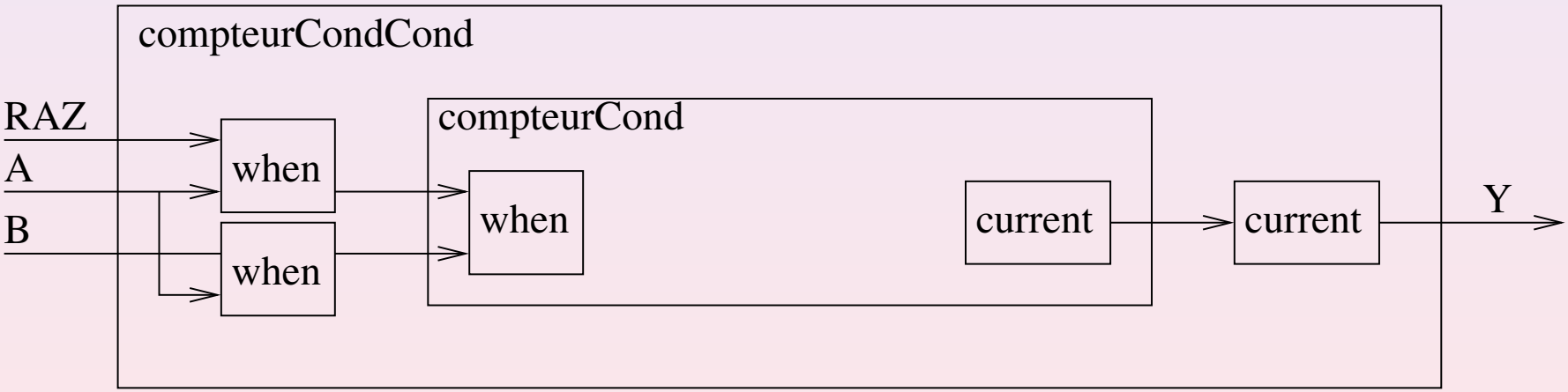
Execution de compteurCondCond :

/



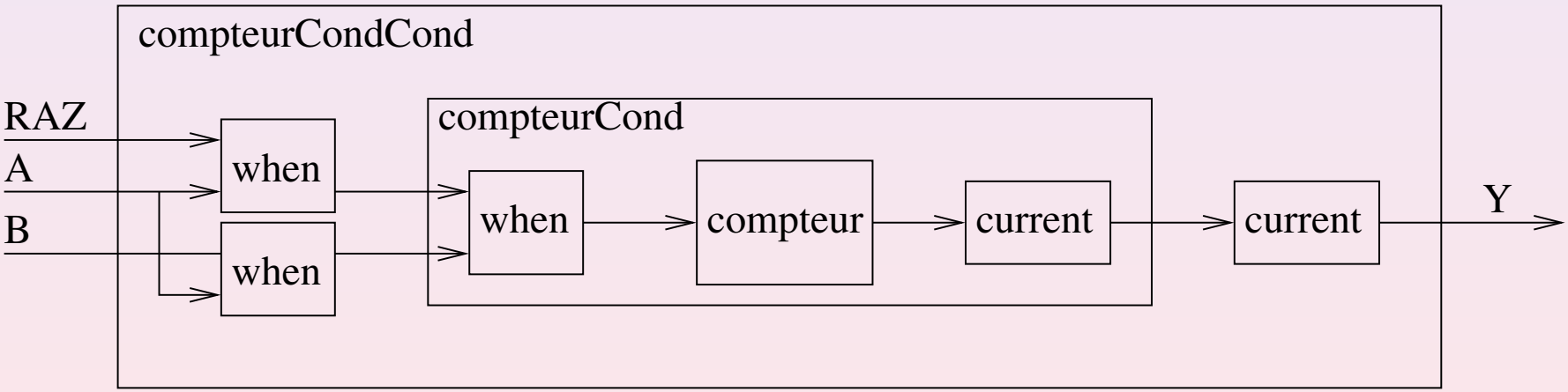
Execution de compteurCondCond :

/-



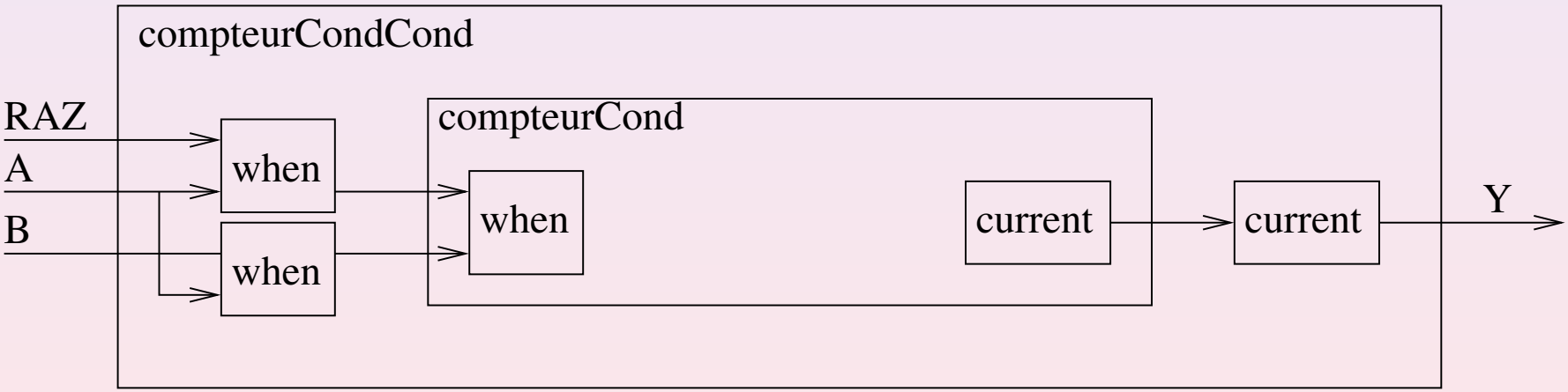
Execution de compteurCondCond :

/-\



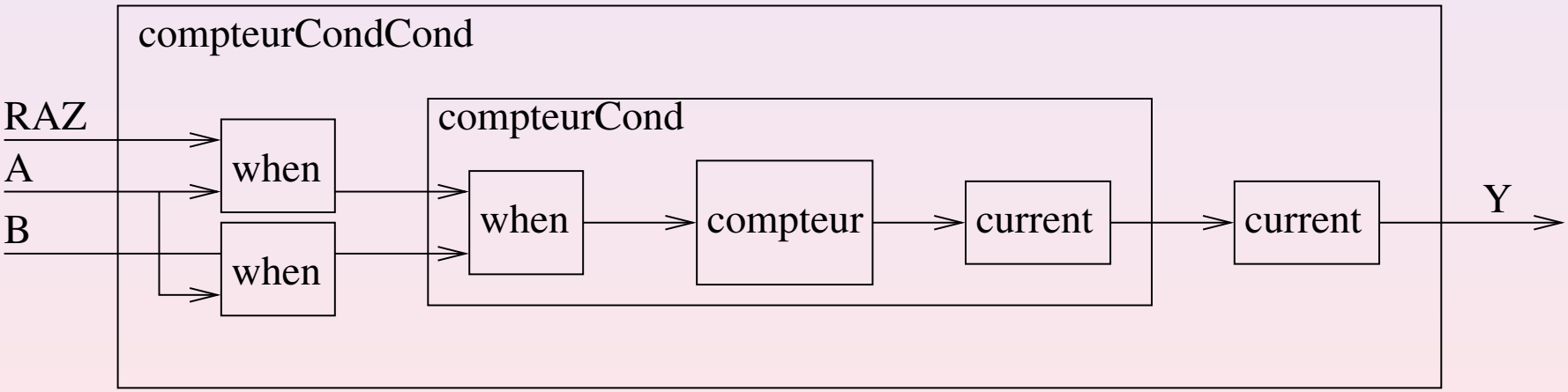
Execution de compteurCondCond :

/-_



Execution de compteurCondCond :

/-_ /

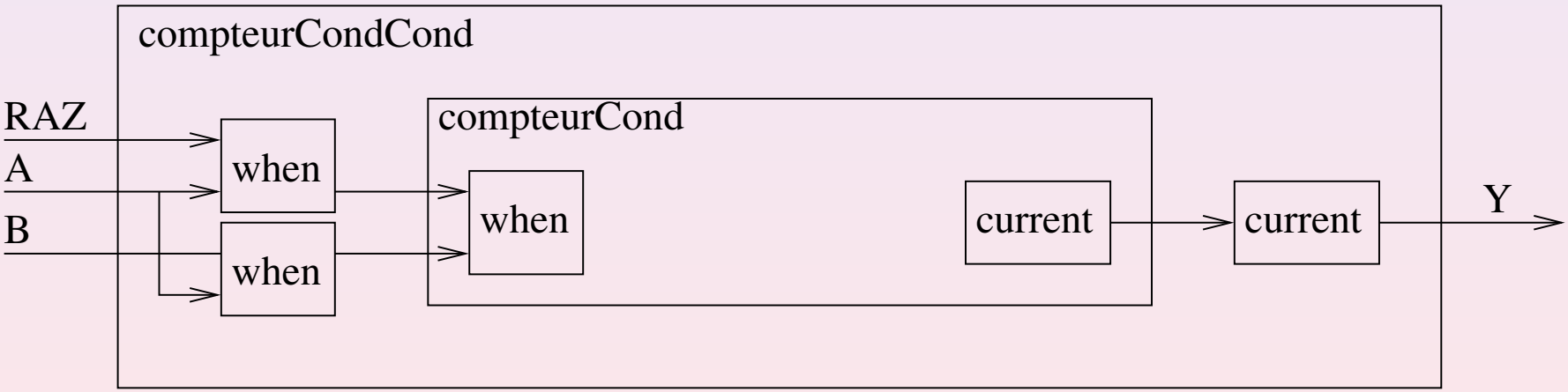


Execution de compteurCondCond :

/-_ /-

Execution de compteurCondCond :

/-_ /-\

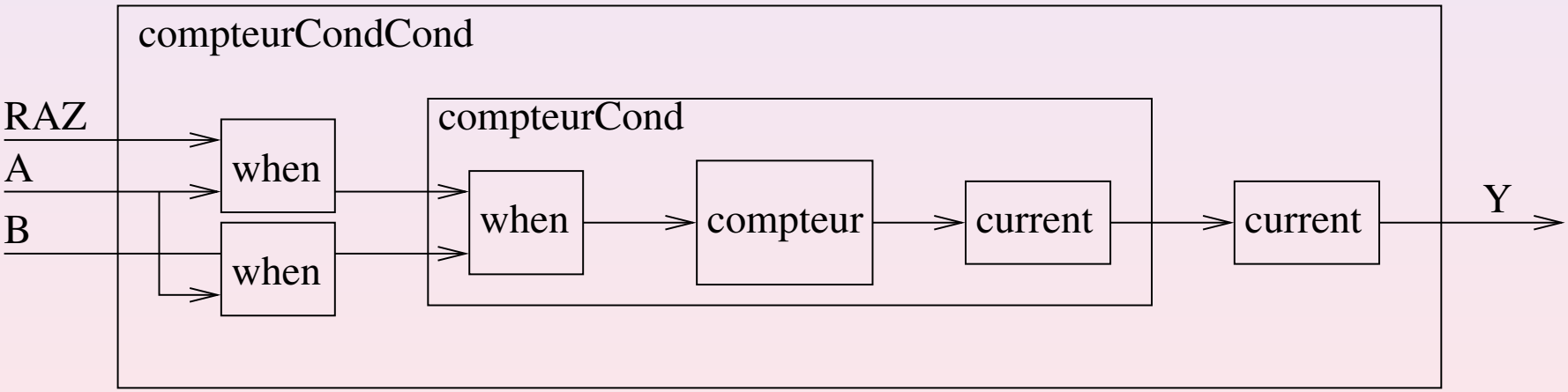


Execution de compteurCondCond :

/-_ /-_

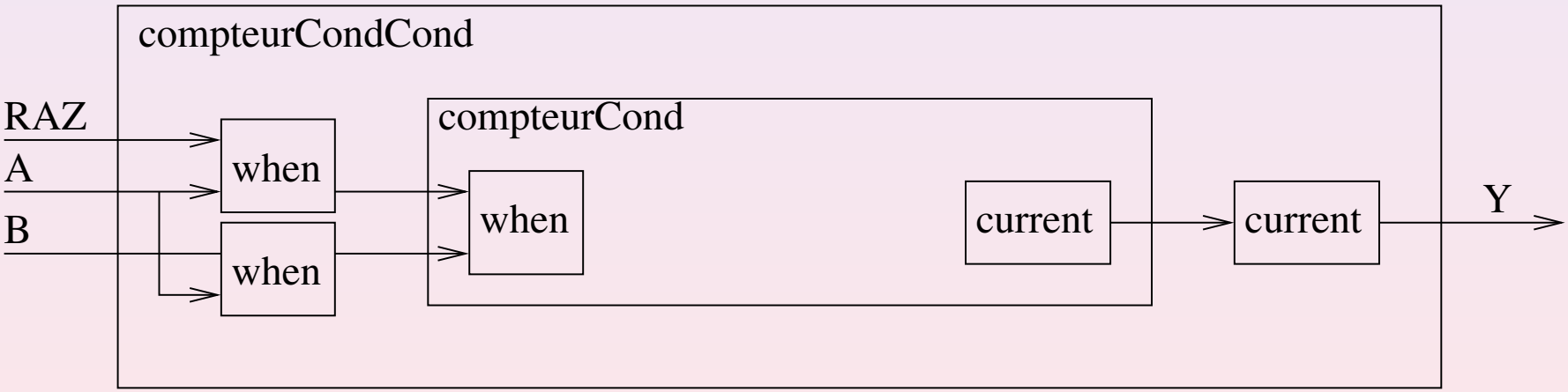
Execution de compteurCondCond :

/-_ /-_ /



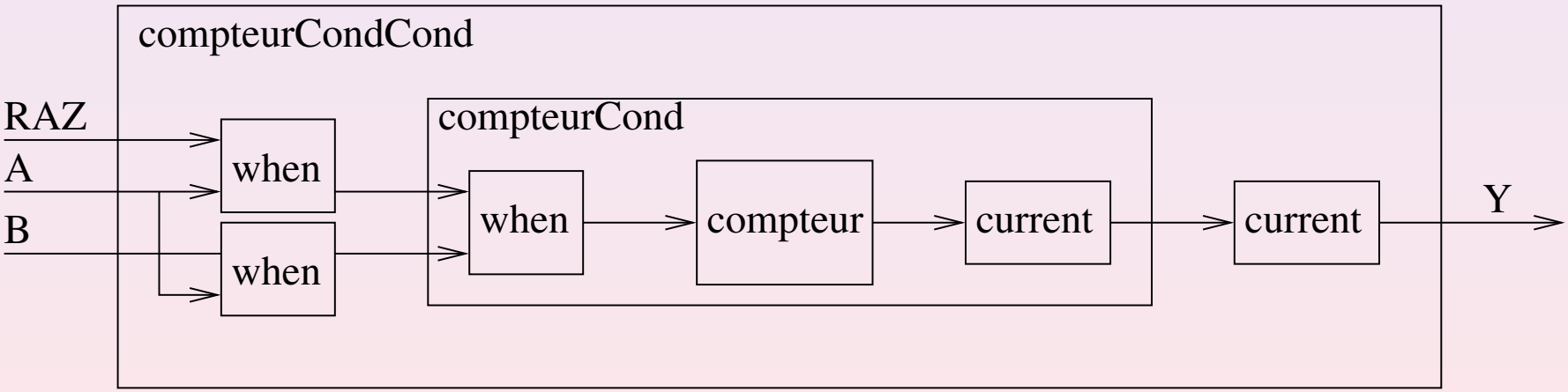
Execution de compteurCondCond :

/-_ /-_ /-



Execution de compteurCondCond :

/-_ /-_ /-\



La notion d'assertion

Les assertions donnent la possibilité au concepteur d'expliciter les hypothèses faites sur l'environnement et/ou sur le programme lui-même.
Ceci permet :

- d'optimiser la compilation
- la vérification de propriétés sous conditions
- de simplifier la conception des programmes

assert : prend en entrée un flot booléen qui exprime l'hypothèse sensée être vrai à chaque instant de l'exécution.

Exemple

Exemple

`assert(not(X and Y));`
affirme que les flots booléens X et Y ne doivent jamais être vrais simultanément.

`assert(true -> not(X and pre(X)));`
affirme que le flot booléen X ne transporte jamais deux valeurs vraies consécutives.

Exercice, une chambre à échos numérique

Il s'agit d'écrire un effet numérique temps réel pour un logiciel de music. L'effet doit simuler une chambre d'échos et compter le nombre de saturations que le signal de sortie à pu subir.
C'est un effet mono il y a donc une entrée de type réel pour le son d'entrée et deux sorties, une de type réel pour le son de sortie (i.e le son d'entrée plus son écho) et une autre de type entier qui renvoie le nombre de saturations qu'il y a eu depuis la mise en marche de l'effect.

La fréquence d'échantillonnage du son est de 1000Hz, on fait l'hypothèse que le signal d'entrée est compris entre -1 et 1 *(utiliser assert)*. L'écho a un delai de 5ms (ce qui correspond à 5 échantillons) et un rapport de 50% .

A chaque fois que le signal de sortie est en dehors des bornes -1 et 1 le compteur de saturation augmente de 1.

Qu'est-ce que la preuve de programme

- La preuve de programme est une manière intelligente de se convaincre qu'un programme se comporte de la bonne façon, qu'il vérifie des propriétés exigées, qu'il n'a pas de bugs.

Qu'est-ce que la preuve de programme

- La preuve de programme est une manière intelligente de se convaincre qu'un programme se comporte de la bonne façon, qu'il vérifie des propriétés exigées, qu'il n'a pas de bugs.
- Le test rassure *"jusqu'ici ça marche"*,
  la preuve assure *"quelque soit la situation ça marchera"*.

Historique

- Cela commence avec le travail des logiciens qui veulent modéliser le raisonnement mathématique (depuis les années 300 après JC avec Aristote jusqu'à aujourd'huis avec les travaux de Hilbert, Gödel et bien d'autres dans les années 30) $\Rightarrow$ logique propositionnelle, logique du premier ordre, théorie de la démonstration, etc.

Historique

- Cela commence avec le travail des logiciens qui veulent modéliser le raisonnement mathématique (depuis les années 300 après JC avec Aristote jusqu'à aujourd'huis avec les travaux de Hilbert, Gödel et bien d'autres dans les années 30) $\Rightarrow$ logique propositionnelle, logique du premier ordre, théorie de la démonstration, etc.

- Ceci conduit avec l'invention des ordinateurs à des systèmes non-humains capablent de tenir des raisonnements mathématiques (Prolog, les systèmes experts, les systèmes d'inférences en général). Puis rapidement à des programmes capables de faire des raisonnements sur des programmes, la methode B, le model checking, etc. Et donc à déceler les bugs dans les programmes sans avoir à faire de longues et coûteuses campagnes de tests.

Les preuves par raisonnement

Les premiers prouveurs travaillent par raisonnement (par exemple Prolog) le problème c'est que les possibilités de raisonnements sont si riches que le prouveur passe son temps à chercher sans jamais trouver, ou en tout cas pas souvent, ou sur des choses très très simples.

Et pour cause, *les programmes informatiques font parties des objets les plus complexes des mathématiques.*

Preuves par vérification exhaustive des états possibles

Pour palier à cette complexité et cette richesse de raisonnement trop lourde à assumer (en tout cas pour l'instant), les *model checkers* sont apparus. Les model checkers n'on qu'*une seule manière de raisonner* mais ils le font de la façon la plus efficace possible.

Preuves par vérification exhaustive des états possibles

Pour palier à cette complexité et cette richesse de raisonnement trop lourde à assumer (en tout cas pour l'instant), les *model checkers* sont apparus. Les model checkers n'on qu'*une seule manière de raisonner* mais ils le font de la façon la plus efficace possible.

Ils font une vérification exhaustive de tous les états possibles du programme à un instant donné. Ils tiennent alors le raisonnement suivant :

- Si c'est vrai dans tous les états possibles du programme alors ce sera vrai quelque soit son exécution et aussi pendant toute la durée de celle-ci.

Preuves par vérification exhaustive des états possibles

Pour palier à cette complexité et cette richesse de raisonnement trop lourde à assumer (en tout cas pour l'instant), les *model checkers* sont apparus. Les model checkers n'on qu'*une seule manière de raisonner* mais ils le font de la façon la plus efficace possible.

Ils font une vérification exhaustive de tous les états possibles du programme à un instant donné. Ils tiennent alors le raisonnement suivant :

- Si c'est vrai dans tous les états possibles du programme alors ce sera vrai quelque soit son exécution et aussi pendant toute la durée de celle-ci.

Problème : si le nombre d'états est infini ou trop grand la vérification est impossible.

La vérification formelle en Lustre

Pour faire la vérification d'un programme, il faut :

1. une description du programme compréhensible par un prouveur (comme sous forme d'automate par exemple)
2. une logique pour décrire les propriétés à prouver.

[2] Ceci pour la raison que le langage Lustre est en fait une logique

La vérification formelle en Lustre

Pour faire la vérification d'un programme, il faut :

1. une description du programme compréhensible par un prouveur (comme sous forme d'automate par exemple)
2. une logique pour décrire les propriétés à prouver.

Pour Lustre il existe un model checker Lesar qui accepte directement les programmes dans la syntaxe Lustre, mais encore mieux, la logique permettant d'écrire les propriétés à satisfaire est aussi du langage Lustre[2] !

[2] Ceci pour la raison que le langage Lustre est en fait une logique

Rappel d'automate

- Un automate est un graphe orienté qui permet de décrire les états possibles d'un système ainsi que son évolution. Dans le cas d'un programme les états possibles correspondent à toutes les combinaisons de valeurs possibles des variables du programme.

Rappel d'automate

- Un automate est un graphe orienté qui permet de décrire les états possibles d'un système ainsi que son évolution. Dans le cas d'un programme les états possibles correspondent à toutes les combinaisons de valeurs possibles des variables du programme.
- Un programme Lustre définie un automate finie car son espace mémoire est fini (en raison de l'impossibilité de l'allocation dynamique). Chaque état correspond à une configuration possible de valeurs de flots d'entrées, internes et sorties pour un instant.

Exemple notE, espace d'états

```
node notE(E :bool)
    returns(S :bool) ;
let
  S=not E ;
tel
```

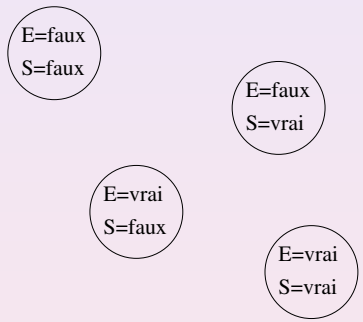


FIG.:

code Lustre et son automate

Exemple notE, automate complet

```
node notE(E :bool)
    returns(S :bool) ;
let
  S=not E ;
tel
```

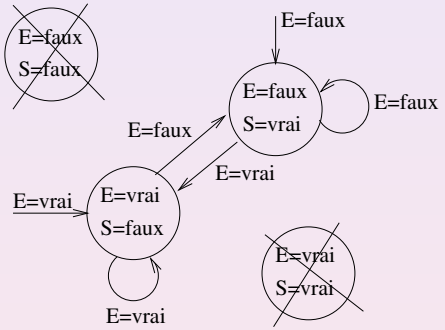


FIG.:

code Lustre et son automate

Exemple notE, automate *à action*

```
node notE(E :bool)
    returns(S :bool) ;
let
  S=not E ;
tel
```

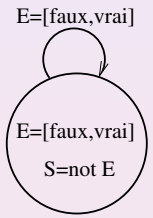


FIG.:

code Lustre et son automate

Exemple oscS, automate complet

```
node oscS(E :bool)
    returns(S :bool) ;
let
  S=false->not pre(S) ;
tel
```

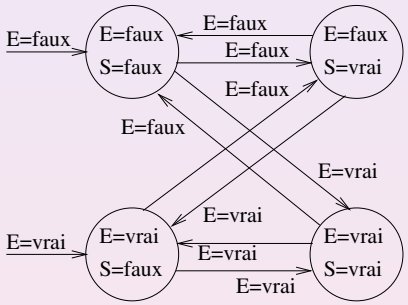


FIG.:

code Lustre et son automate

Exemple oscS, automate *à action*

```
node oscS(E :bool)
    returns(S :bool) ;
let
  S=false->not pre(S) ;
tel
```

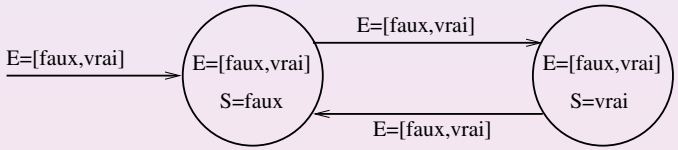


FIG.:

code Lustre et son automate

Automate avec plus d'un pre

- Idée : *inclure l'histoire du programme dans l'état courant.*

```
node osc2S(E :bool)
    returns(S :bool) ;
let
  S=not(false->pre(false->pre(S))) ;
tel
```

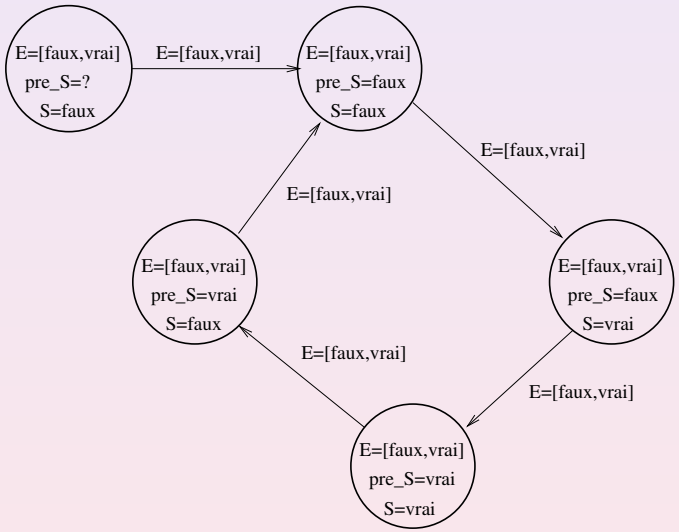


FIG.:

code Lustre et son automate

Safety Logic

En Lustre le langage qui sert à décrire les formules à vérifier est le même que celui qui sert à programmer. Il s'agit de la logique temporelle *SL* pour *Safety Logic*. C'est une logique à temps qualitatif (notion d'ordres entre les événements), capable d'exprimer des propriétés dîtes de *sûreté*.

Un programme Lustre exprime une relation logique/mathématique entre des flots d'entrées et des flots de sorties de types bool, int et/ou real.
Une propriété Lustre exprime une relation logique entre des flots d'entrées de types bool un unique flot de sortie de type bool[3].

[3] En fait on peut travailler sur des flots de n'importe quel type mais dans la plus part des cas Lesar sera incapable de prouver quoique ce soit si les flots ne sont pas booléens

Expréssivité de SL

On ne peut pas tout exprimer en SL, seules les propriétés de sûreté (qu'on ne définira pas précisément ici). Sa définition mathématique est un peu compliqué[4] et n'est pas indispensable à connaître pour l'utilisation de cette logique. On peut sans doute justifier le fait que Lustre ne produit que des propriétés de sûretés de par son point de vue tournée vers le passé.

[4] elle dit très grossièrement que les propriétés de suretés sont les propriétés qui ne peuvent pas faire la distinction entre les exécutions infinis et finis

Une propriétés telle que *"un jour il se produira tel événement"* n'est pas une propriété de sûreté. En revanche une propriété telle que *"un jour il se produira tel événement avant tel autre"* est une propriété de sûreté.

Schéma général de la vérification en Lustre

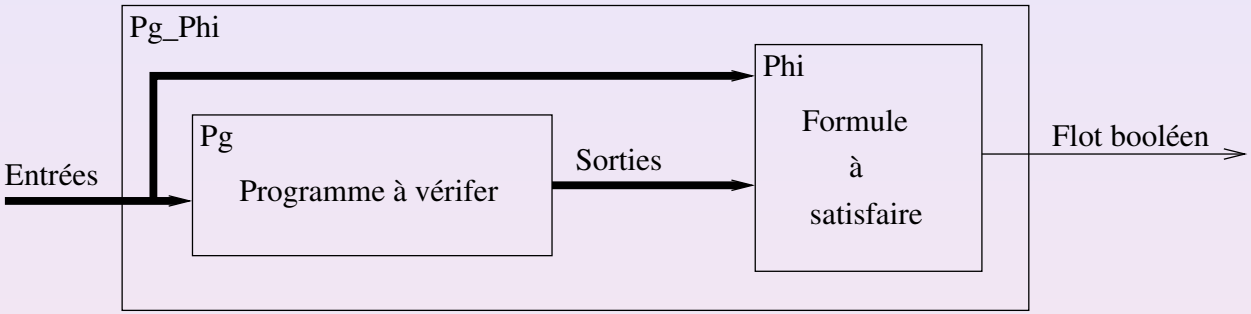


FIG.: *Schéma général d'une vérification de programme en Lustre*

- *Pg* est le programme à vérifier écrit en Lustre
- *Phi* est l'observateur, il définit des relations entre les entrées et les sorties du programme, ces relations doivent être vraie à tout instant. C'est la propriété à démontrer et celle-ci s'écrite également en Lustre
- *Pg_Phi* est le programme à mettre dans le model checker

La satifaction de *Phi*

Phi est satisfaite si et seulement si pour toute entrée E[5] et pour tout instant, le flot booléen B en sortie de Pg_Phi revoie la valeur true.

[5] E représente l'ensemble des flots d'entrées et pas seulement un flot

Exemple

```
node prog1(E :int)returns(S :bool) ;
let
  S=if E=0 then not(pre(S)) else
       if E>0 then true else false ;
tel
```

```
node observateur2(E :int ;S :bool)returns(B :bool) ;
let
  assert(E=0) ;
  B=true -> (S xor pre(S)) ;
tel
```

```
node prog_observé2(E :int)returns(B :bool) ;
let
  B=observateur2(E,prog1(E)) ;
tel
```

Attention ne pas confondre le assert avec le if . then . else dans la formulation de la propriété. Le assert est temporellement global tandis que le if est temporellement local.

Définir des opérateurs temporels complexes

Lustre ne possède qu'un opérateur temporel permettant de voyager dans le temps, le pre, il est très simple (il ne fait que considérer la valeur d'un flot à son instant précédent), pourtant il va permettre de définir toute une variétés d'opérateurs temporels utilisables dans la suite pour formaliser des propriétés temporelles assez complexes.

Exemple, l'opérateur temporel *never*

A chaque instant *never* renvoie vrai si est seulement si le flot d'entrée n'a jamais été vrai.

```
node never(X : bool)
    returns(P : bool) ;
let
  P=if X then false else (true -> pre(P)) ;
tel
```

Exemple, l'opérateur temporel *never*

A chaque instant *never* renvoie vrai si est seulement si le flot d'entrée n'a jamais été vrai.

```
node never(X : bool)
    returns(P : bool) ;
let
  P=if X then false else (true -> pre(P)) ;
tel
```

Exemple

temps	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*...*
X	*faux*	*faux*	*faux*	*vrai*	*faux*	*faux*	*vrai*	*vrai*	*...*
P	*vrai*	*vrai*	*vrai*	*faux*	*faux*	*faux*	*faux*	*faux*	*...*

l'opérateur temporel *since*

L'opérateur *since* renvoie vrai à chaque instant où dans le passé (présent inclu), si une occurence[6] de X est apparu alors elle a été suivi (immédiatement ou avec un certain délai) d'une occurence de Y.

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

[6] sous-entendu occurence d'une valeur vrai

Exemple d'exécution

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

Exemple

temps	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*. . .*
X	*faux*	*faux*	*vrai*	*faux*	*faux*	*faux*	*vrai*	*vrai*	*. . .*
Y	*faux*	*vrai*	*faux*	*faux*	*vrai*	*faux*	*faux*	*vrai*	*. . .*
P	*vrai*	*vrai*	*faux*	*faux*	*vrai*	*vrai*	*faux*	*vrai*	*. . .*

On combine maintenant *since* et *never* pour définir l'opérateur *once_B_from_A_to_C*

On combine maintenant *since* et *never* pour définir l'opérateur *once_B_from_A_to_C*

- Soit la propriété suivante :
  "toute occurrence de A doit être suivie par une occurrence de B avant la prochaine occurrence de C"

On combine maintenant *since* et *never* pour définir l'opérateur *once_B_from_A_to_C*

- Soit la propriété suivante :
  "toute occurrence de A doit être suivie par une occurrence de B avant la prochaine occurrence de C"
- Qui, exprimée au passé donne :
  "à chaque occurrence de C, il faut que A ne se soit jamais produit, ou si A s'est produit, il faut que B se soit produit depuis la dernière occurrence de A"

On combine maintenant *since* et *never* pour définir l'opérateur *once_B_from_A_to_C*

- Soit la propriété suivante :
  "toute occurrence de A doit être suivie par une occurrence de B avant la prochaine occurrence de C"
- Qui, exprimée au passé donne :
  "à chaque occurrence de C, il faut que A ne se soit jamais produit, ou si A s'est produit, il faut que B se soit produit depuis la dernière occurrence de A"

```
node once_B_from_A_to_C(A,B,C : bool)
    returns(P : bool);
let
  P = if C then (never(A) or since(A,B)) else true;
tel
```