Langages pour le temps réel

ENSEEIHT/SN – 3A – parcours E&L

Frédéric Boniol

Frederic.boniol@onera.fr

2022-2023

Synchronous languages

ENSEEIHT 3A – parcours E&L

2022/2023

Frédéric Boniol (ONERA)
frederic.boniol@onera.fr
Damien Guidolin-Pina (Onera), Pierre-Julien Chaine (Airbus)

Summary: Objective of the course

- Lecturer
  - Frédéric Boniol
  - ONERA (Office National d'Etudes et de Recherches Aérospatiale)
  - Teaching and research domain =
    - Embedded systems
    - Real-time programming
    - Formal verification (formal methods for embedded systems)

- Objectives
  - Introduction to Synchronous Languages
    - What are the main principles of the synchronous languages
  - Focus on LUSTRE

    => a formal data flow synchronous language for programming control systems

Summary: Plan of the course

- Theoretical part: introduction to real-time systems
  - Lecture 1.
    - Brief overview of "real-time" and "embedded systems"
    - Differences between "synchronous" and "asynchronous languages"
  - Lecture 2.
    - LUSTRE ("single-clock")
  - Lecture 3.
    - LUSTRE ("multi-clock")
    - Formal aspects of LUSTRE: clock calculus
  - Lecture 4.
    - Formal verification of LUSTRE programs
    - Exercises

Course mark:
- 1h30 exam
- All documents are allowed

- Practical part (Thomas Beck and Pierre-Julien Chaine):
  - 3 sessions on
    - Small LUSTRE programs
    - Lego Robot

Synchronous languages

Lecture 1:
Embedded and real-time systems

ENSEEIHT 3A – parcours E&L
2022/2023

Frédéric Boniol (ONERA)
frederic.boniol@onera.fr

1. About embedded systems…

1.1. Some general definitions:

- What is an "embedded" system?
- What is a "real-time" system?

1.2. An exemple: the flight control system

1.3. Generalisation

1.4. Question: do we need specific languages for programming embedded software?

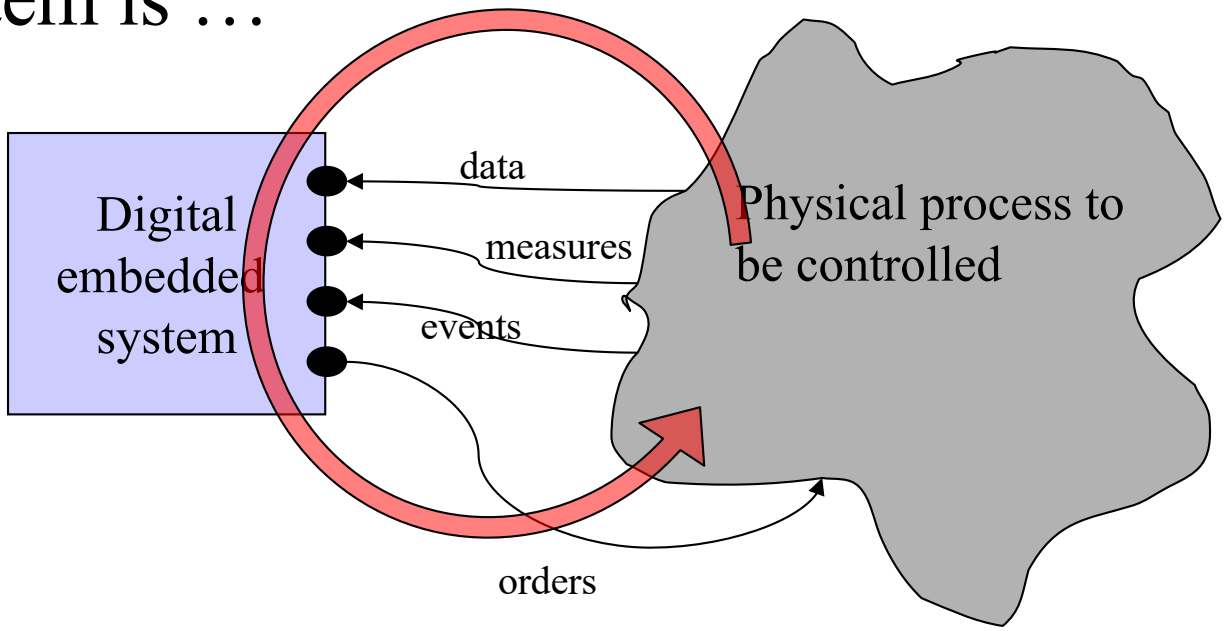
What is a digital embedded system?

- Information processing system embedded into a larger product [Pr. Marwedel, Dortmund Univ]
  - Composed of
    - Software components
    - Running on Hardware Components
  - Integrated with the physical process to be controlled

⇒ The main technical problem is
  - managing time and concurrency
    in the computational part of the embedded system.

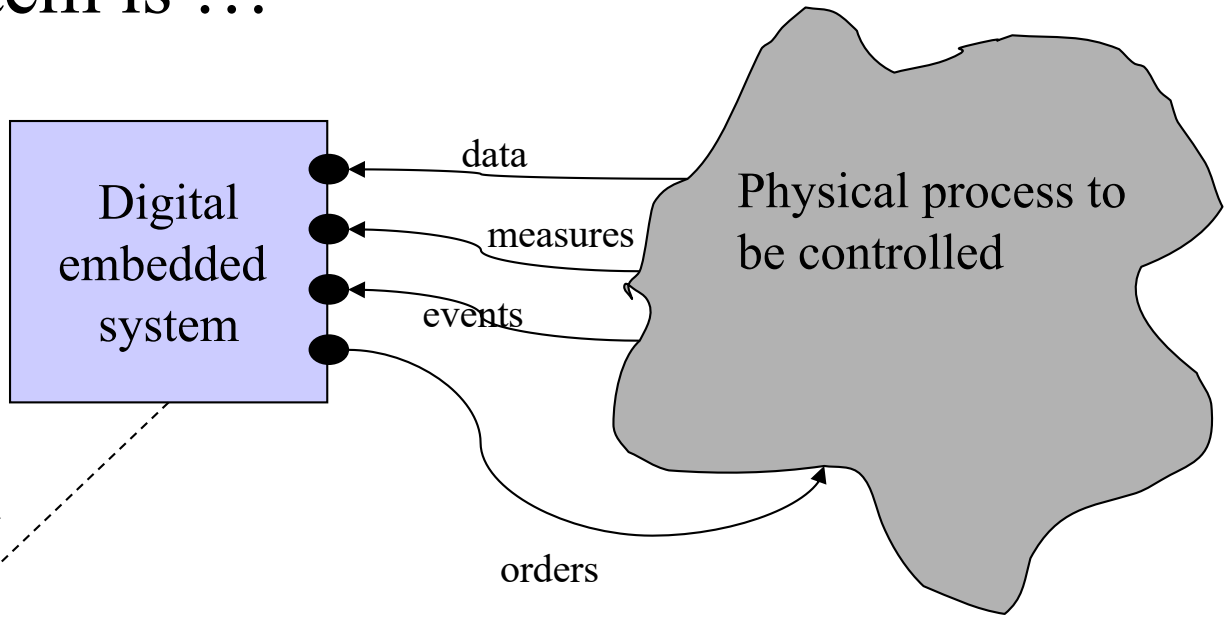
An embedded system is …

… strongly connected in a closed loop with the process to be controlled



An embedded system is …

… strongly connected in a closed loop with the process to be controlled

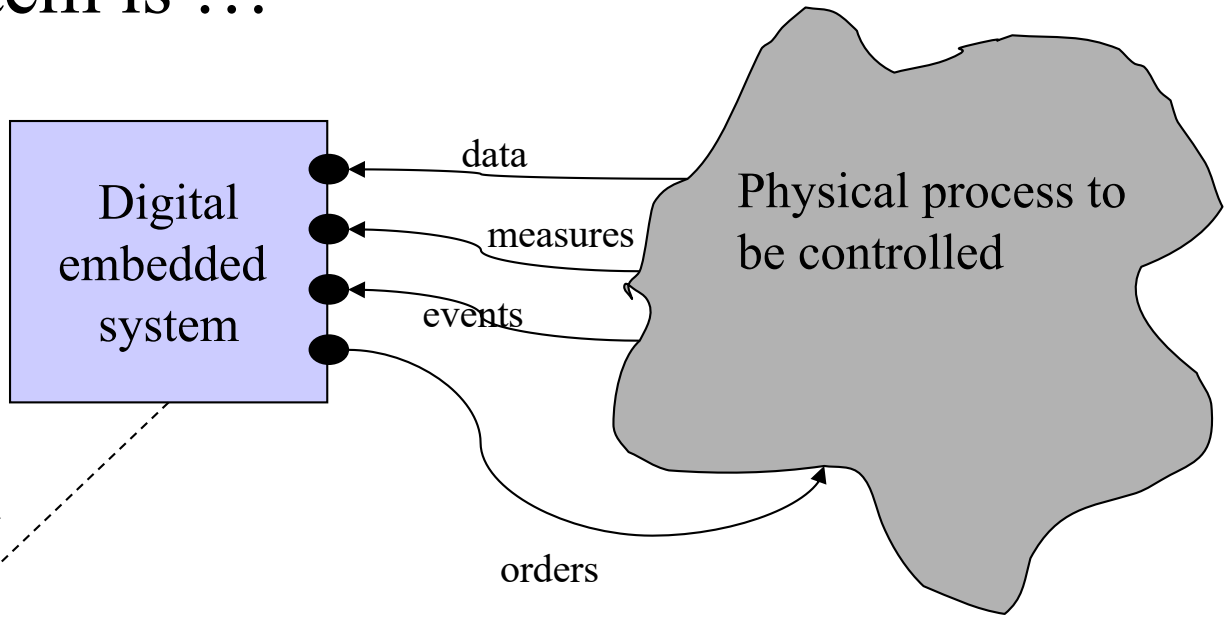


… may be implemented by

- an integrated circuit (ASIC, FPGA)
- a sequential SW on single-core processor
- **a multi-threaded SW on a single-core or multi-core processor**
- ...

An embedded system is …

… strongly connected in a closed loop with the process to be controlled



… may be implemented by

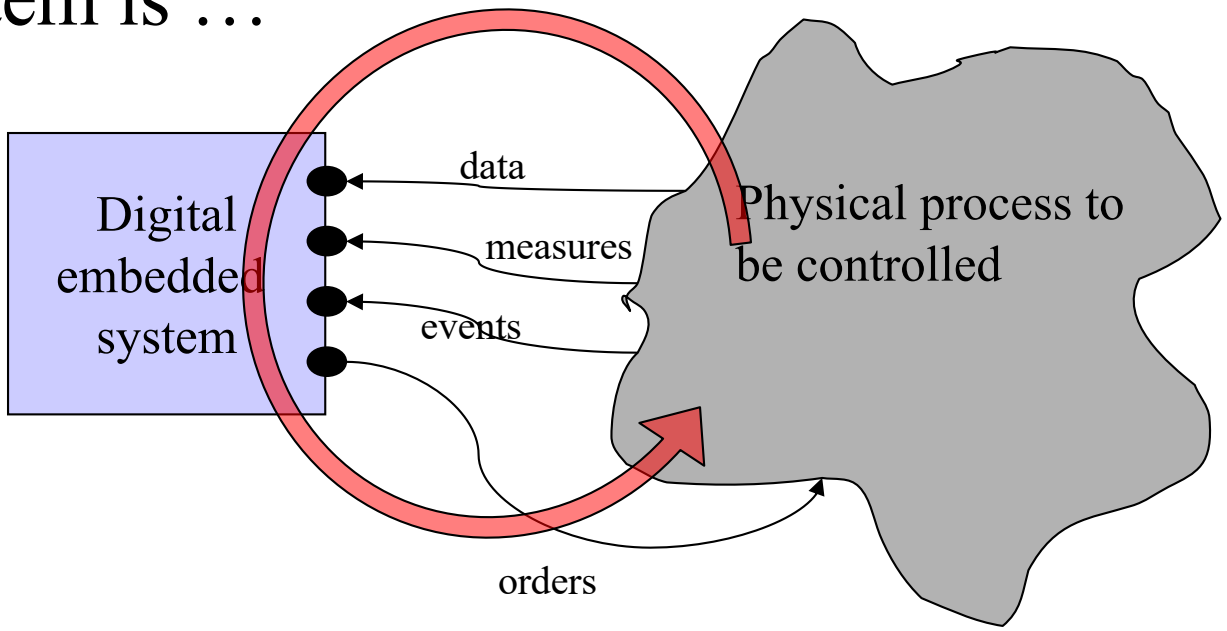
- an integrated circuit (ASIC, FPGA)
- a sequential SW on single-core processor
- a **multi-threaded SW on a single-core or multi-core processor**
- …

⇒ Characteristics of Embedded Systems

- Must be dependable
- Must be efficient (energy, weight, cost, etc.)
- **Must meet real-time constraints**

An embedded system is …

… a real time system

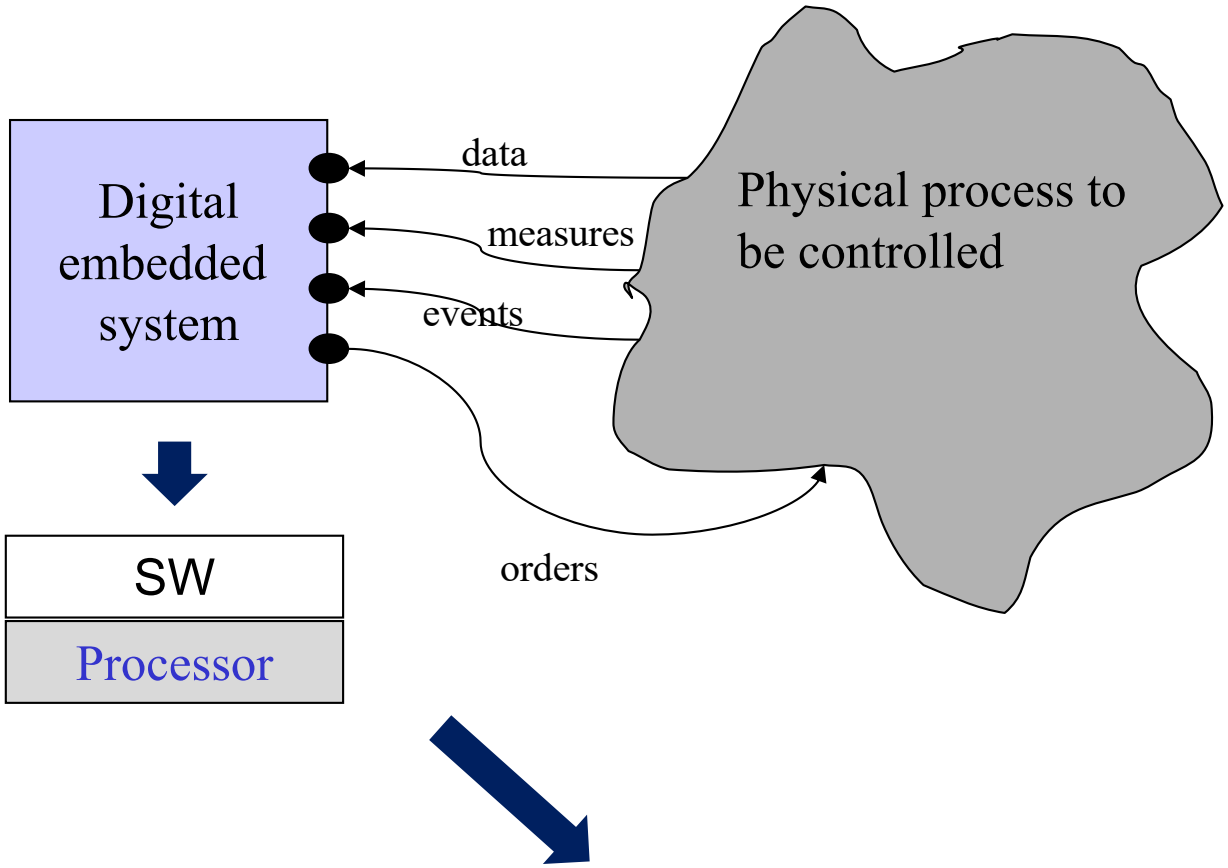


⇒ Characteristics of Embedded Systems

- Must be dependable
- Must be efficient (energy, weight, cost, etc.)
- **Must meet real-time constraints**

Two types of timing constraints:
- Periodicity of control loop.
- Latency of the control loop.

The question…



Implemented by a mutli-threaded SW on a single-core or multi-core processor.

Two types of timing constraints:
- Periodicity of control loop.
- Latency of the control loop.

Question: what kind of languages / mechanisms do we need for programming the embedded systems?

1. About embedded systems…

1.1. Some general definitions:

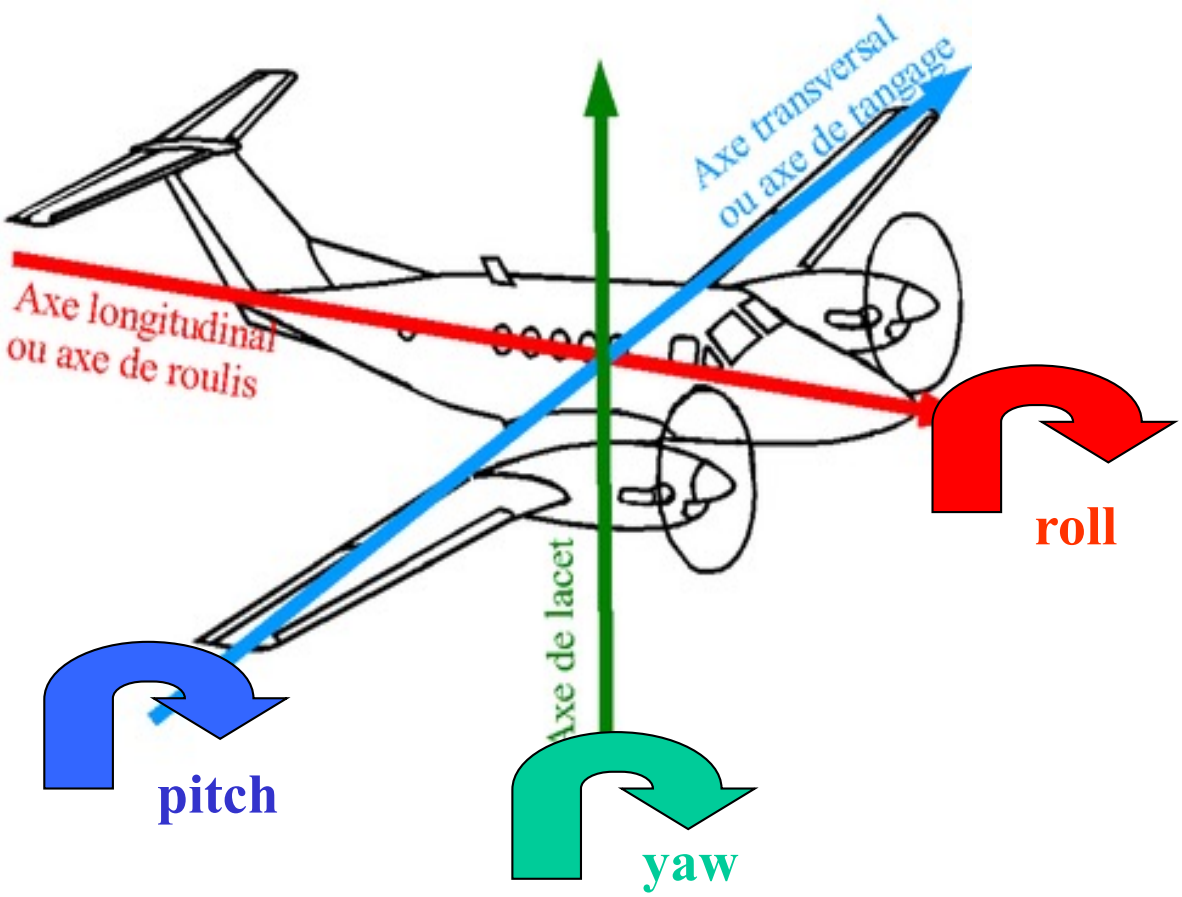
- What is an "embedded" system?
- What is a "real-time" system?

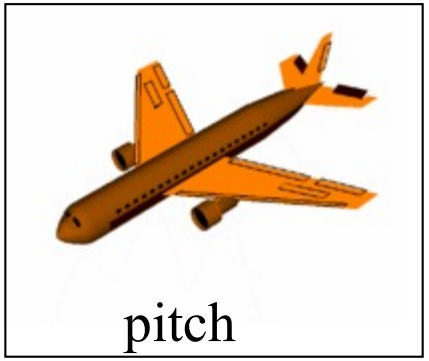
1.2. An exemple: the flight control system

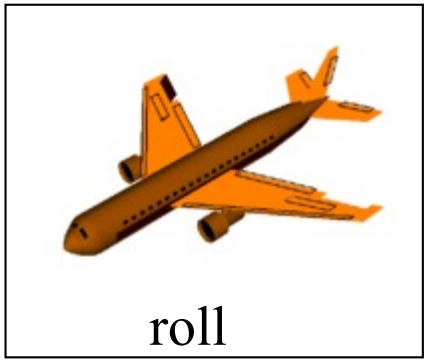
1.3. Generalisation

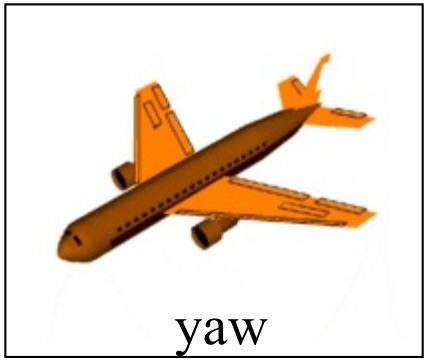
1.4. Question: do we need specific languages for programming embedded software?

Flight control

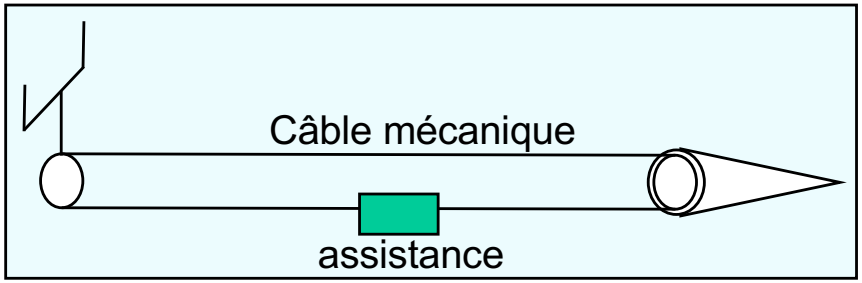




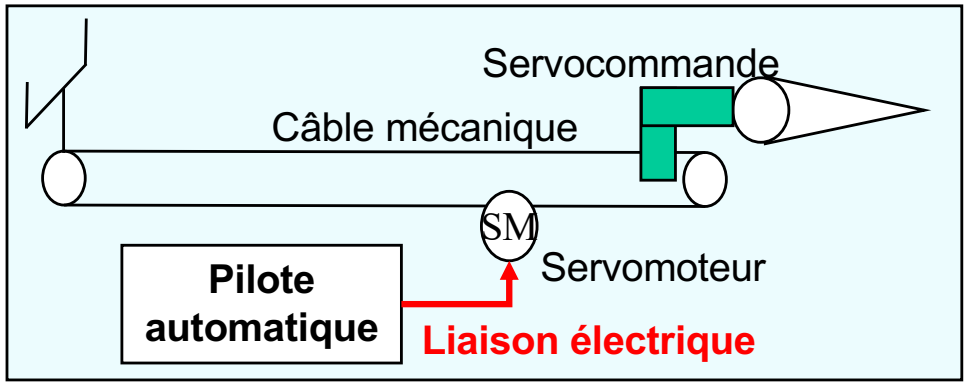




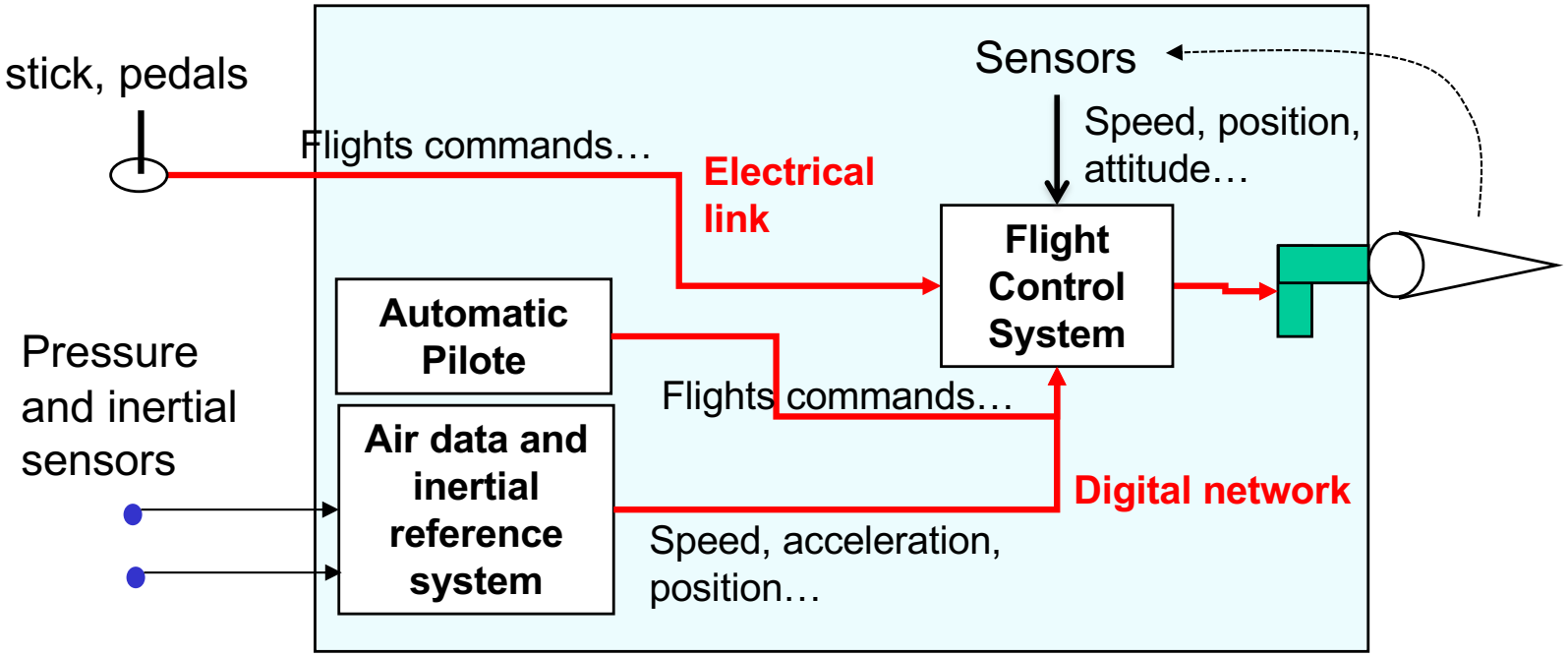
=> The Flight Control System - Evolution



V0: fully mechanical flight control system



V1: mechnical flight control system + digital flight guidance (before A320)

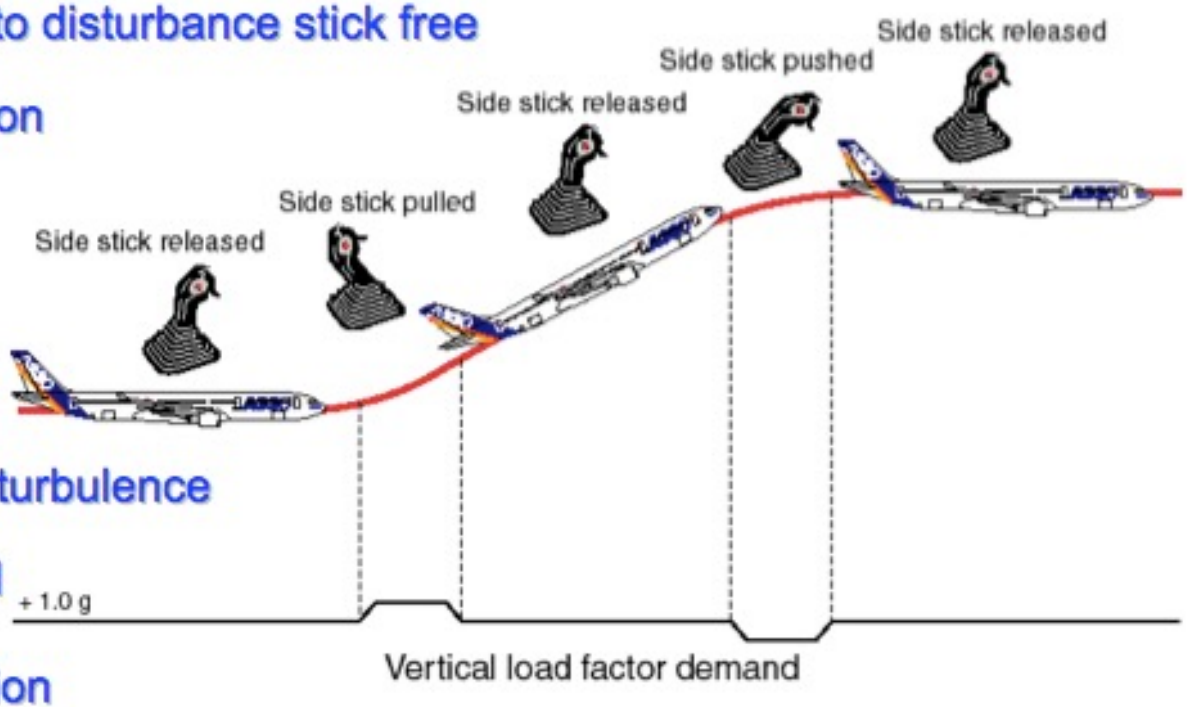


V2: digital flight control system (A320 …)

=> The Flight Control System – V2

Role of the flight control system

- ✓ Automatic pitch trim
- ✓ A/C response (almost) unaffected by speed, weight or centre of gravity location
- ✓ Bank angle resistance to disturbance stick free
- ✓ Efficient turn coordination
- ✓ Dutch roll damping
- ✓ Side-slip minimization
- ✓Accurate flying
- ✓ Passengers comfort in turbulence
- ✓ Reduced pilot workload
- ✓ Flight envelope protection



=> The Flight Control System – V2

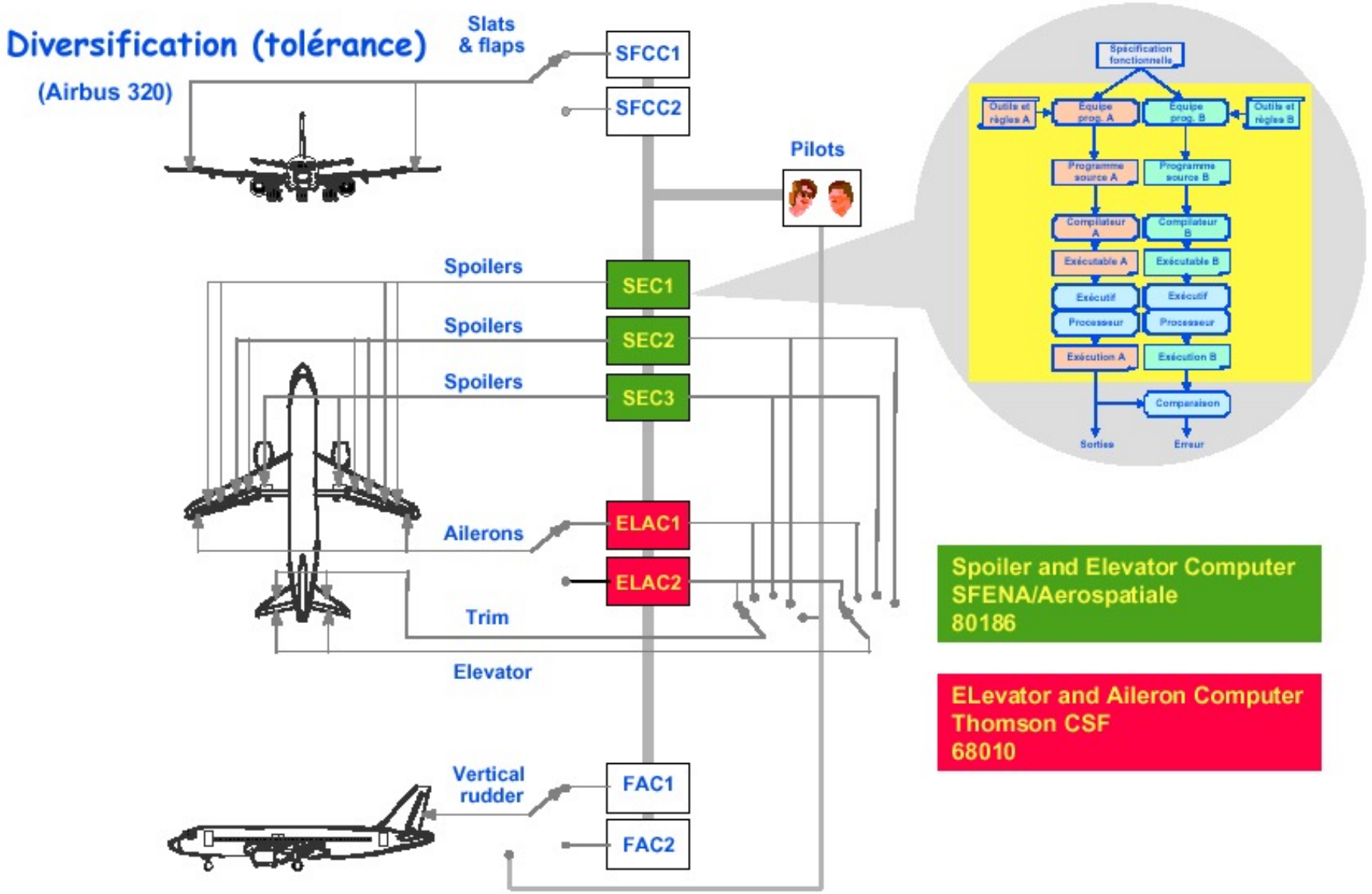
Role of the avionic software

- To monitor the health of the system,
- To reconfigure the system in case of abnormal behaviour

=> Example: A320 Flight control system

=> The Flight Control System – A320

- The A320 Flight control system



1. About embedded systems…

1.1. Some general definitions:

- What is an "embedded" system?
- What is a "real-time" system?

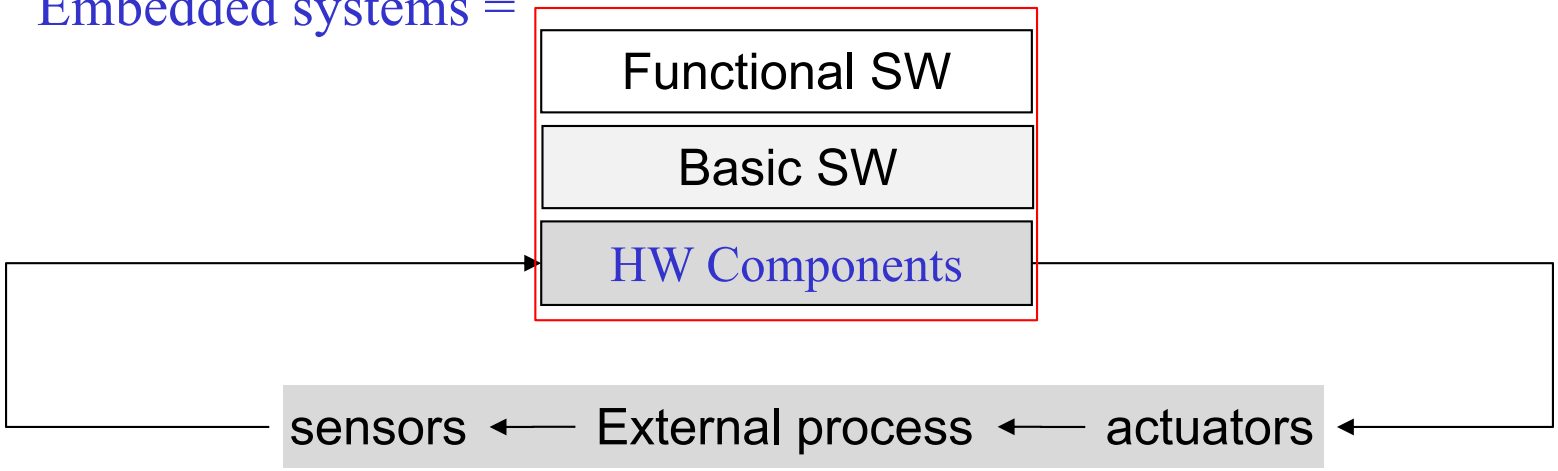
1.2. An exemple: the flight control system

1.3. Generalisation

1.4. Question: do we need specific languages for programming embedded software?

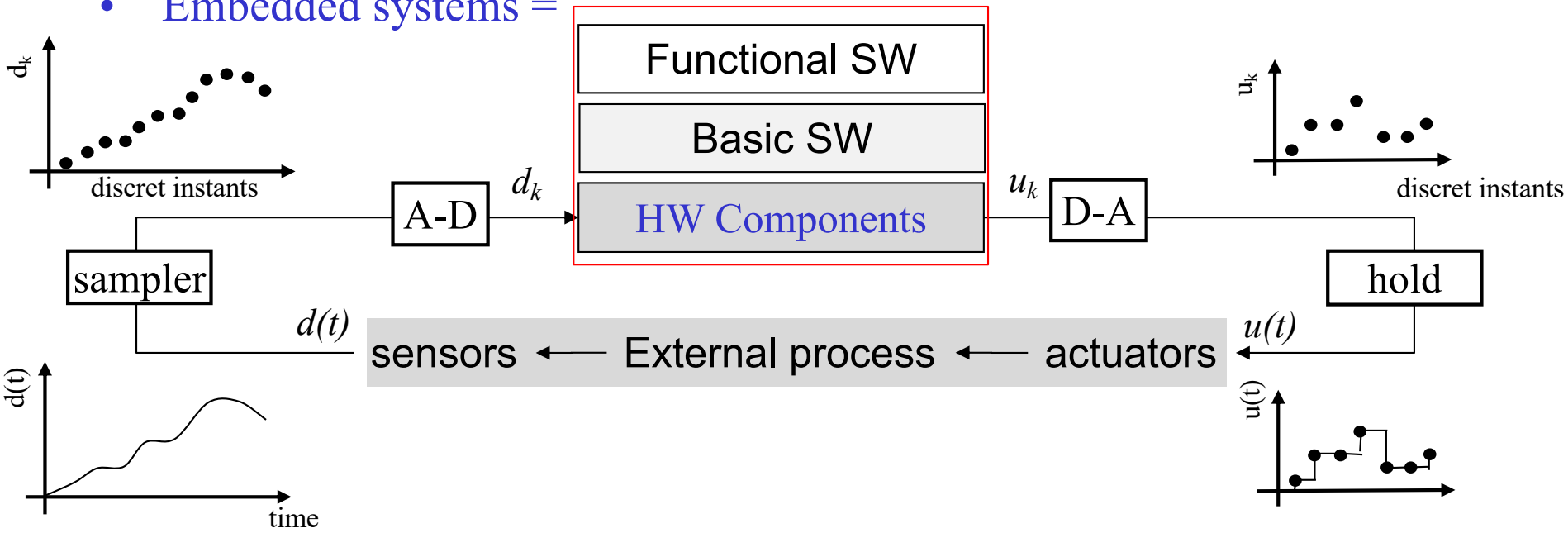
Generalisation

- Embedded systems =



Generalisation

- Embedded systems =



⇒ Systèmes échantillonnés !

⇒ Rythmés par une horloge globale !

⇒ Manipulant des « flots » (d_k, u_k, …)

1. About embedded systems…

1.1. Some general definitions:
- What is an "embedded" system?
- What is a "real-time" system?

1.2. An exemple: the flight control system

1.3. Generalisation

1.4. Question: do we need specific languages for programming the functional part of embedded systems?

Generalisation

⇒ Question:

- **Do we need any specific languages** for programming functional SW?

⇒ It depends on the complexity of the SW

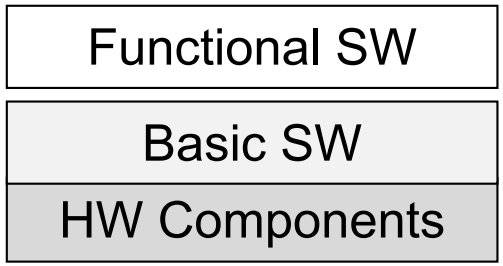
⇒ Yes for today (complex) systems

⇒ No for previous (simple) systems

⇒ Yes for safety-critical systems:

⇒Need of formal proofs

⇒ What kind of languages?

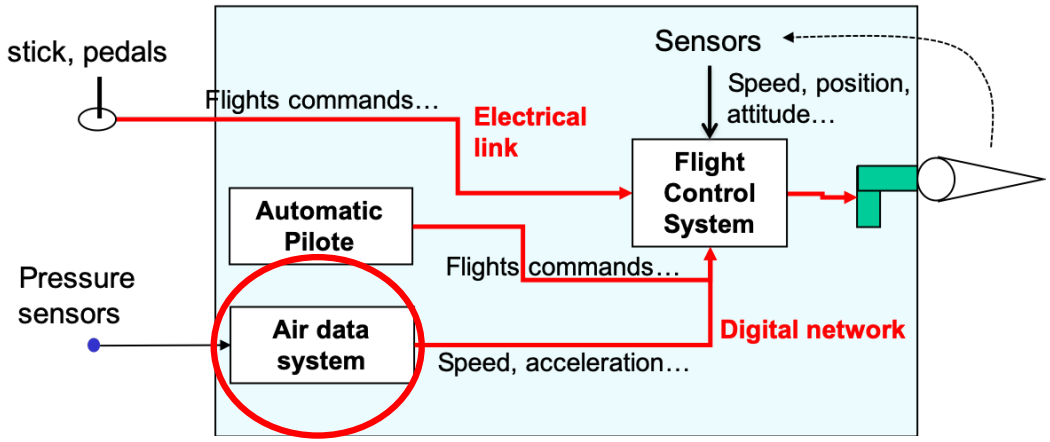


2. Question: do we need specific languages for programming functional SW

Example 1: single-period system

Example 1: AD system (Air Data)

- Computes the Mach number and the altitude of an aircraft from pressure information

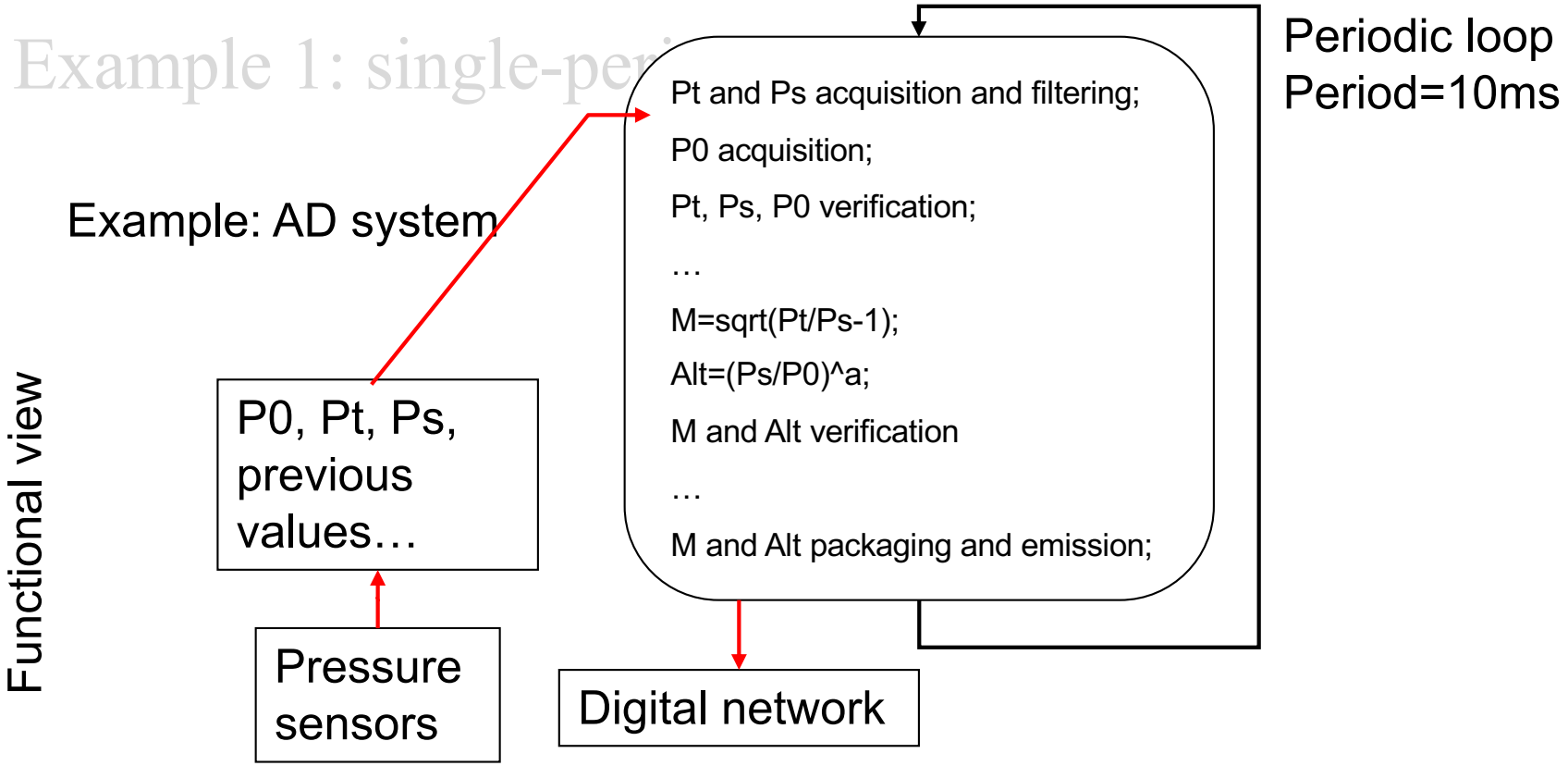


⇒ Simple system (single-period system)

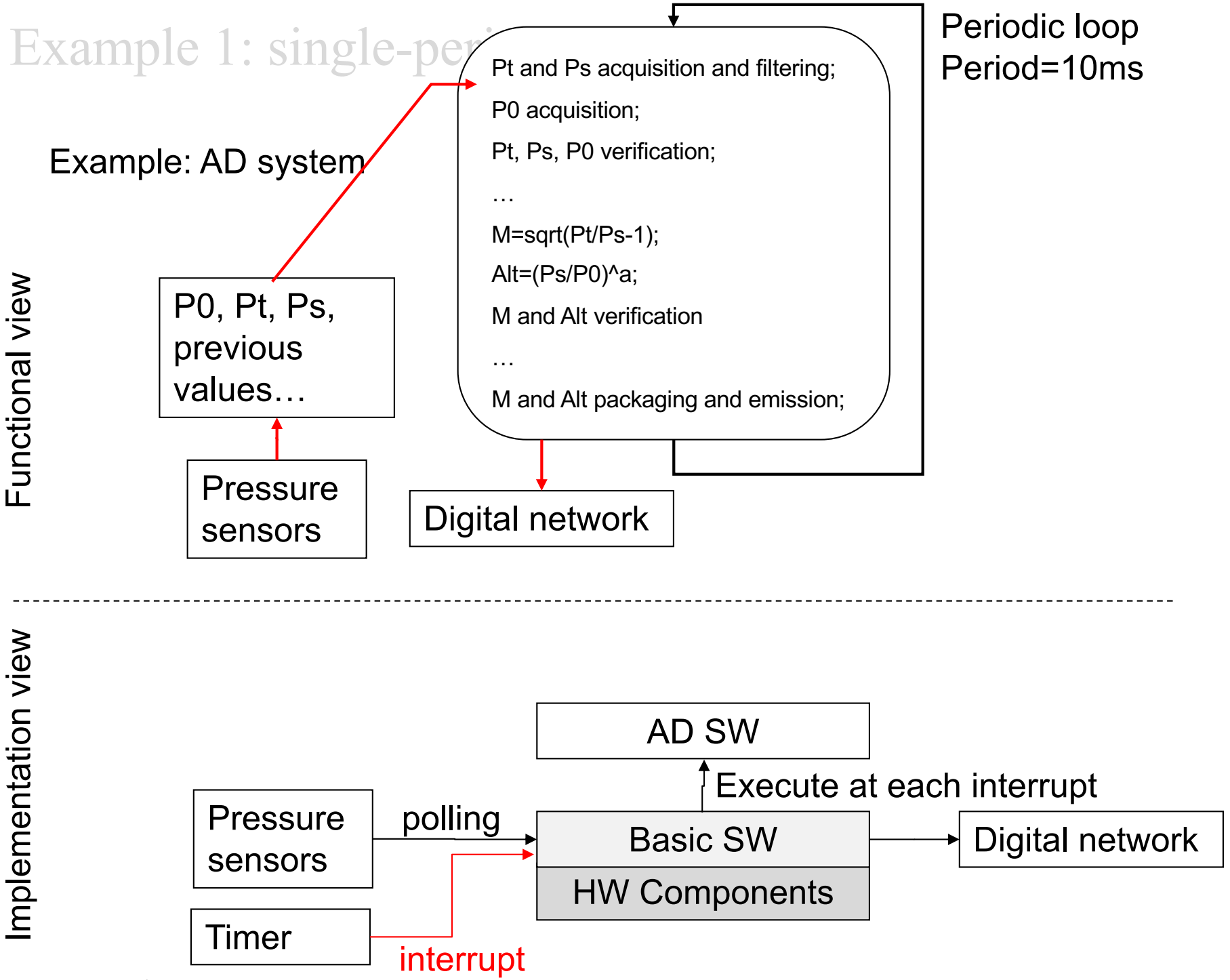
- composed of only one sequential code
- periodically executed
- only one period (10 ms)
- on a single-core HW platform

(Case of the first generation of avionic systems (up to A320))

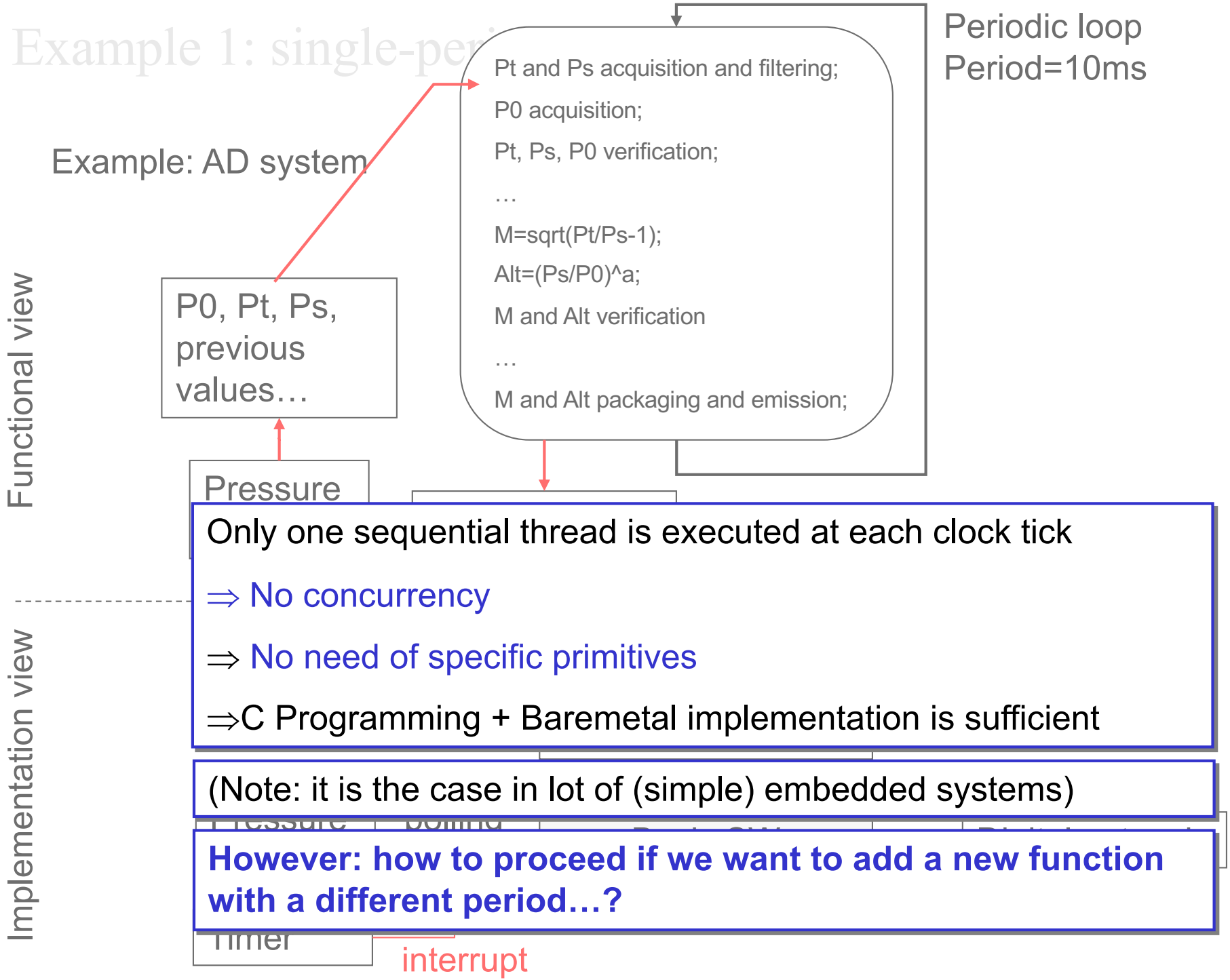
Example 1: single-peri



Example 1: single-peri



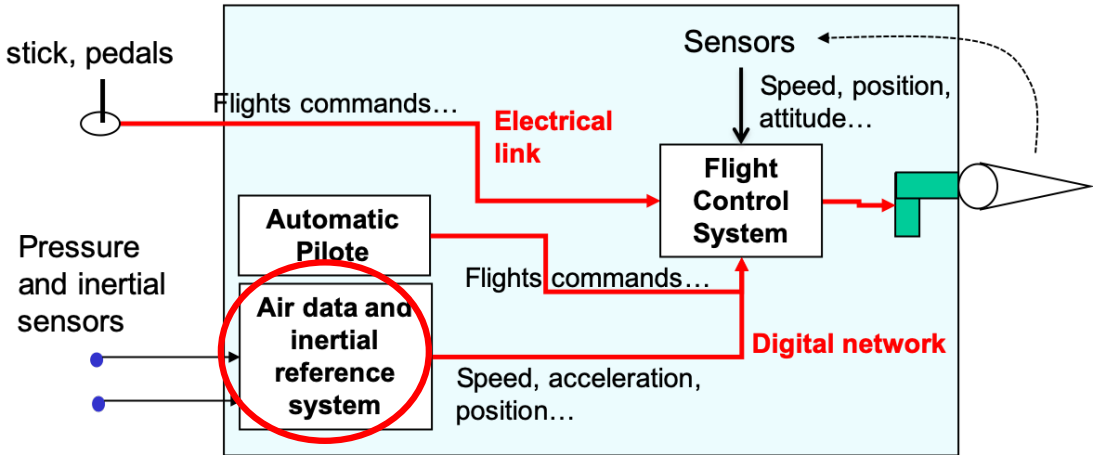
Example 1: single-peri



Example 2: multi-period system

Example 2: ADIR system (Air Data and Inertial)

- Computes the Mach number and the altitude of an aircraft from pressure information
- Computes the position of the aircraft from gyroscope information
- Check correctness



⇒ Multi-threaded system

- composed of several functional (sequential) activities
- each activity is periodic, but they may have different period

=> they can ask for processor at the same time

=> Concurrency problem

Generalization

- Today generation of embedded software (A350, military aircraft, space vehicles…)
  - composed of
    - several periodic activities (triggered by periodic clocks)
    - several aperiodic activities (triggered by external events)

⇒ Question:
- how to specify / program these activities?

⇒ Two ways:

⇒ Asynchronous programming =
- C programming
- Real-Time Operating System

⇒ Synchronous programming =
- Specific and simpler languages
- Compiler to C implantation

3. Synchronous versus Asynchronous programming

- Challenges of asynchronous programming
- Principle of synchronous programming

Challenges of asynchronous programming

What is « asynchronism »

Asynchronism =

The threads do not share the same « time »

=> No global time

=> Durations of instructions are not defined in the semantics of the languages

=> Durations are undefined

Benefit:

Fit well with the concrete behaviour of the HW architectures.

Problem:

Concurrency is not deterministic

⇒A program can behave differently with the same inputs

⇒Increases the difficulty in mastering the behaviour of the programs

Challenges of asynchronous programming

Example:

```
Signal X : integer ;

X = 0;
[
   X = 1;
   X = 2;
||
   Y = X+1;
]
```

Asynchronism
=> Each branch run in parallel at its own pace

Asynchronous semantics => ,

several interleaving behaviour

=> several results for Y: 1, 2 or 3

⇒ Non-deterministic execution

Challenges of asynchronous programming

Conclusion:

Asynchronism is source of complexity and difficulty.

It requires control mecanisms to prevent non-determinisic behaviours

$\Rightarrow$ Increase in the complexity of programming languages and of the programs!

The synchronous « idea »

Idea:

Simplify the programming model

=> The two synchronous simplifications

- All the threads of the system share the same « time »
- Execution time = 0

Remark:

Similar to physicists' approaches

=> approach by simplification (i.e., to ignore unnecessary details)

=> Here: abstraction of real execution time!

=> Benefit:

– determinism

⇒ Drawback: not compliant with the real life

⇒ Model to be confirmed by comparison with the concrete execution

The synchronous « idea »

Example:

```
Signal X : integer combine with + ;

X = 0;
[
   X = 1;
   X = 2;
||
   Y = X+1;
]
```

Synchronism
=> All the branches run simultaneously and instantaneously

Synchronous semantics => ,

Only one interleaving behaviour

=> Only one result for Y = 4

=> Single assignation

=> Deterministic semantics

The synchronous « idea »

Benefit example:

```
Signal X : integer combine with + ;
X = 1;
X = 2;
```

congruent to

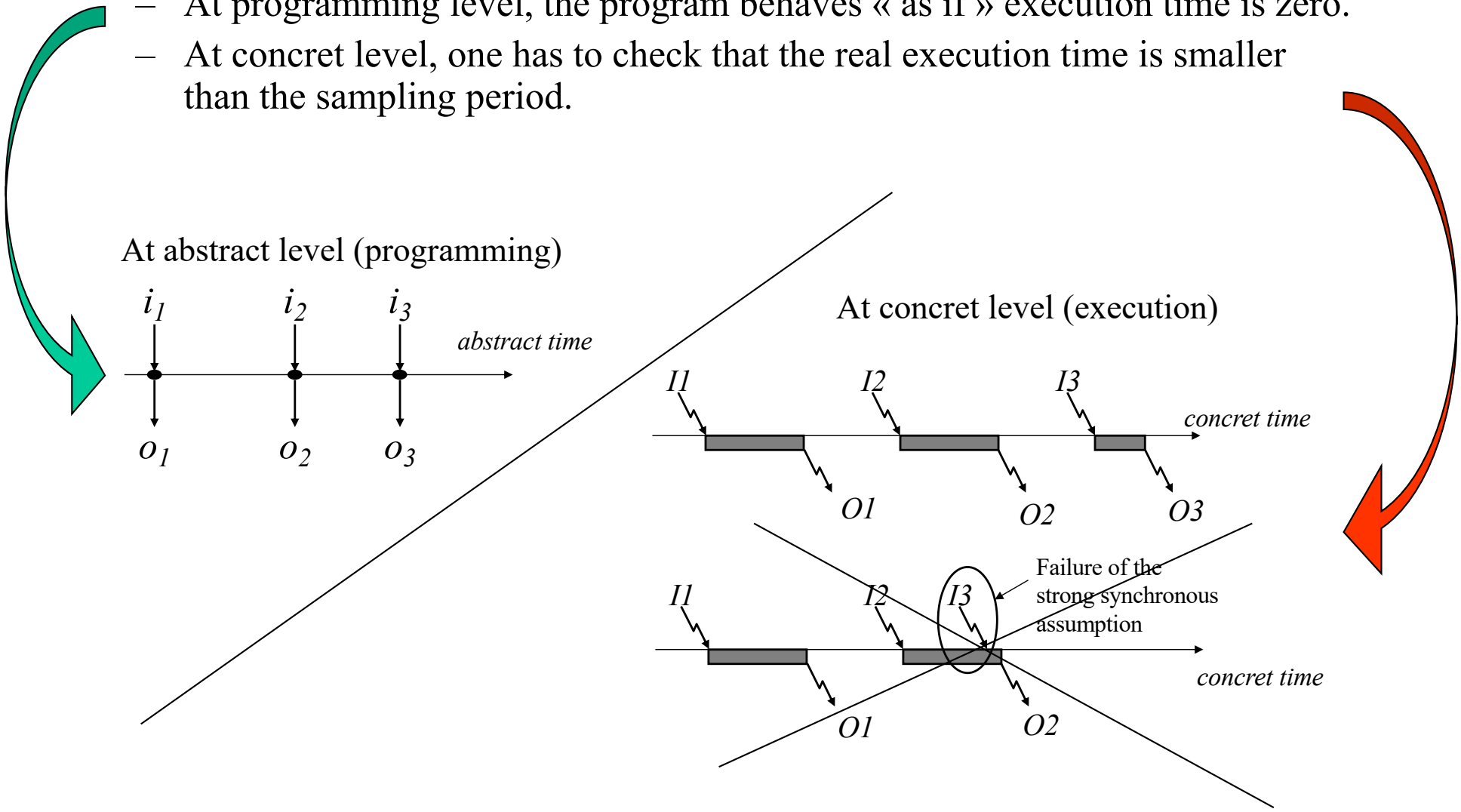
```
Var X : integer combine with + ;
X = 3;
```

=> It allows code optimisation at compile time

The synchronous « idea »

Strong synchronous assumption: *execution time = 0*

- At programming level, the program behaves « as if » execution time is zéro.
- At concret level, one has to check that the real execution time is smaller than the sampling period.



Two main synchronous languages

- LUSTRE:
  - Data flow programming
  - Equational style
- ESTEREL:
  - Event flow programming
  - Imperative style

End of lecture 1

$\Rightarrow$Next lecture: LUSTRE (simplified version)

Synchronous languages

Lecture 2 : Lustre basic

Frédéric Boniol

ONERA - 2, av. E. Belin - 31055 Toulouse

frederic.boniol@onera.fr

1. Le langage Lustre : généralités

Motivation :

Permettre la programmation « naturelle » et « sûre » de systèmes de contrôle commande.

Moyen

Techniques de programmation proches des descriptions traditionnelles utilisées par les ingénieurs de ces domaines :

=> blocs diagrammes et flots de données

=> systèmes échantillonnés

=> LUSTRE

langage de programmation formel défini en 1985 par P. Caspi et N. Halbwachs à Grenoble (Vérimag)

- Distribution commerciale : SCADE - Esterel Technologie
- Utilisations industrielles : Airbus, Dassault Aviation, Thales, Schneider Electric…

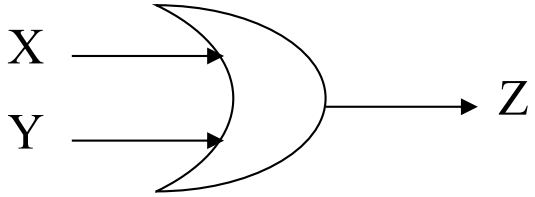
1. Le langage Lustre : généralités

- Caractéristiques générales du langage
  - Objets central : les « flots » :
    - portée locale, entrée, sortie
    - typé
    - Définis par des « équations »
  - Principe data-flow
    - Résolution d'une équation uniquement lorsque tous ses flots « d'entrée » sont présents et calculés
    - Effet = rendre présent et évalué le flot de sortie de l'équation
  - Principe synchrone
    - Portée temporelle des calculs = l'instant courant
    - Accès aux valeurs des instants précédents par mémorisation du passé

1. Le langage Lustre : exemple introductif

Exemple :

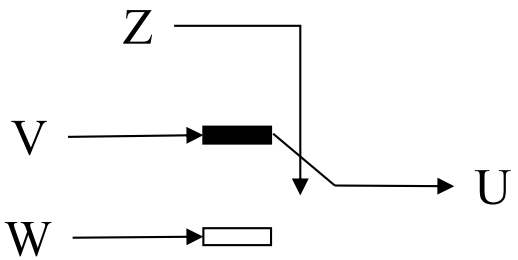
Une porte logique




```
Z = X or Y;
```

pour tout $n \geq 0$, $Z_n = X_n \; or \; Y_n$

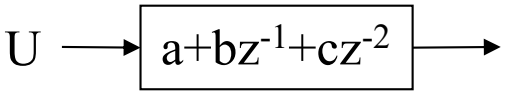
Un switch




```
U = if Z then V
    else W;
```

pour tout $n \geq 0$, $U_n = if \; Z_n \; then \; V_n \; else \; W_n$

Un filtre



pour tout $n \geq 2$, $O_n = aU_n + bU_{n-1} + cU_{n-2}$

```
O = a*U
    + b*pre(U)
    + c*pre(pre(U));
```

(attention : équation incorrectement initialisée)

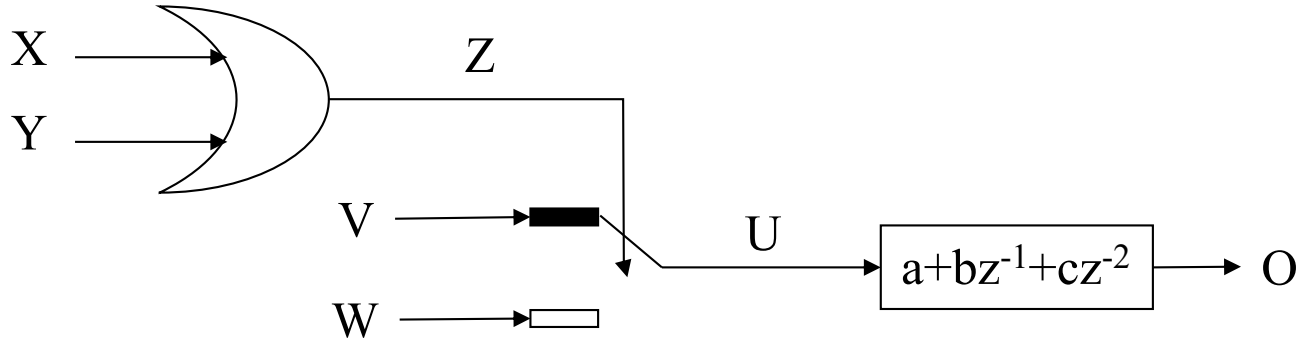
1. Le langage Lustre : généralités…

Généralisation :

– description d'un système en terme de suites de valeurs échantillonnées : les flots de données

=> un système = un réseau d'opérateurs opérant sur des flots de données

Exemple :




```
Z = X or Y;
U = if Z then V else W;
O = a * U + B * pre(U) + c * pre(pre(U));
```

(attention : programme incorrectement initialisé)

2. Le langage Lustre…

Généralisation :

- Notion de flot de données :

  X = suite de valeurs X_n pour n≥0 (flot infini de valeurs)

  X_n = valeur de X à l'instant *n* ($n^{ième}$ top d'horloge)

  Exemples :

  - `1` est le flot infini (`1,1,1,1,1,`…)
  - `true` = (`vrai`, `vrai`, `vrai`, `vrai`,…)

- Chaque flot interne ou de sortie est défini par une équation

  `O = X op Y`

  calculant O_n en fonction de X_n et Y_n (au même instant)

  => `op` opère à chaque instant sur les valeurs de l'instant courant (application point à point)

=> un programme LUSTRE =

- un ensemble d'équations
- qui modélise un processus rythmé par une horloge logique (pas forcément régulière) : à chaque top de cette horloge, le processus LUSTRE calcule des flots de sorties en fonction des valeurs des flots d'entrée à cet instant

2. Le langage Lustre…

Un programme LUSTRE = un ensemble de nœuds…

```
[déclaration de types et de fonctions externes]
node nom (déclaration des flots d'entrée)
returns (déclaration des flots de sortie)
[var déclaration des flots locaux]
let
   [assertions]
   système d'équations définissant une et une seule fois les flots
   locaux et de sortie en fonction d'eux mêmes et des flots d'entrée
tel.

[autres nœuds]
```

Déclaration de flots :

```
NomDuFlot : TypeDuFlot;
```

Flots constant :

```
const NomDuFlot : TypeDuFlot = valeur ;
```

Les types :

- les types de bases : `int`, `bool`, `real`
- les tableaux : `int^3`, `real^5^2`…

2. Le langage Lustre…

Les équations

– une équation définit un flot interne ou de sortie en fonction de flots internes, d'entrée ou de sortie

$$\begin{cases} X = Y + Z \\ Z = U \end{cases}$$

signifie

pour tout $n \geq 0$, $X_n = Y_n + Z_n$ *et* $Z_n = U_n$

=> principe de substitution : une équation définit une égalité mathématique, et non une affectation informatique => un flot peut être remplacé par sa définition dans toutes les équations du nœud

$$\begin{cases} X = Y + Z \\ Z = U \end{cases} \quad \text{équivalent à} \quad \begin{cases} X = Y + U \\ Z = U \end{cases}$$

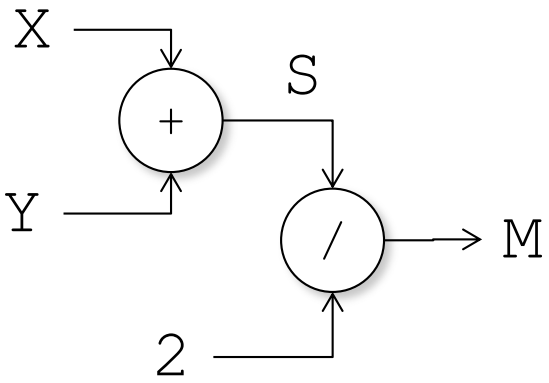
=> Principe data-flow : les équations n'ont pas d'ordre

$$\begin{cases} X = Y + Z \\ Z = U \end{cases} \quad \text{équivalent à} \quad \begin{cases} Z = U \\ X = Y + Z \end{cases}$$

2. Le langage Lustre…

- Exemple : calcul d'une moyenne de deux valeurs

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



Une équation pour chaque sortie et chaque variable locale

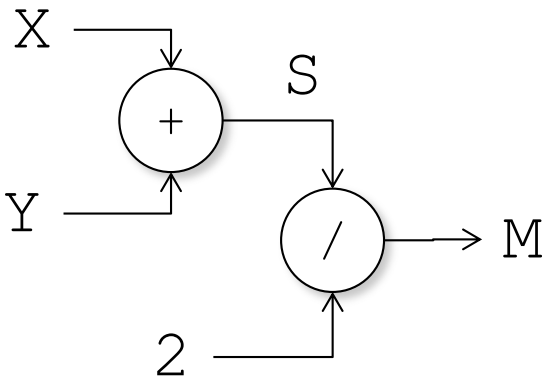
Interprétation du temps : pour tout $n \geq 0$,

- $S_n = X_n + Y_n$
- $M_n = S_n / 2$

2. Le langage Lustre…

- Exemple (suite) : équivalent à (par principe de substitution)

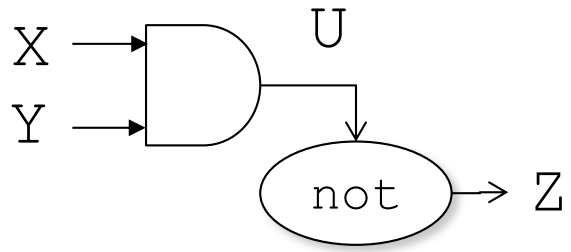
```
node Moyenne (X, Y : int)
returns (M : int);
let
   M = (X + Y) / 2 ;
tel.
```



2. Le langage Lustre…

- Exemple :

```
node Nand (X, Y : bool) returns (Z : bool)
var U : bool;
let
   U = X and Y;
   Z = not U;
tel.
```



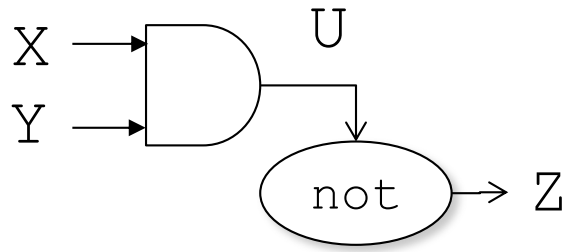
	tick	0	1	2	3	4	5	6	7
inputs	X	true	true	false	false	true	true	false	false
inputs	Z	false	true	false	false	true	false	false	true
local	U	false	true	false	false	true	false	false	false
output	Z	true	false	true	true	false	true	true	true

Équivalent à (par principe de substitution) =>

2. Le langage Lustre…

- Exemple :

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



	tick	0	1	2	3	4	5	6	7
inputs	X	true	true	false	false	true	true	false	false
inputs	Z	false	true	false	false	true	false	false	true
output	Z	true	false	true	true	false	true	true	true

3. Les opérateurs

Opérateurs classiques

Les opérateurs arythmétiques :

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

Les opérateurs logiques :

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

Les opérateurs de comparaison :

- `=`, `<>`, `<`, `>`, `<=`, `>=`

Les opérateurs de contrôle :

- `if . then . else`

Opérateurs temporels :

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

`->` (suivi de) : opérateur permettant d'initialiser un flot

`when` : opérateur de sous-échantillonnage

`current` : opérateur de sur-échantillonnage

3.1. pre et ->

L'opérateur `pre` (précédent)

Permet de mémoriser la valeur précédente d'un flot ou d'un ensemble de flots

Soit

X le flot $(X_0, X_1, \ldots, X_n, \ldots)$

alors

`pre(X)` est le flot $(\text{nil}, X_0, X_1, \ldots, X_n, \ldots)$

Par extension, l'équation

```
(Y, Y') = pre(X, X')
```

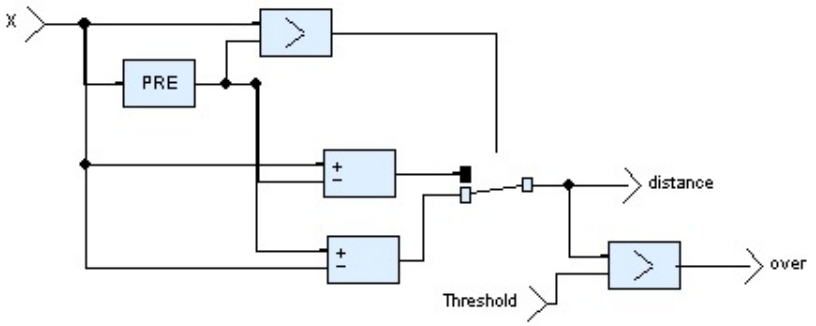
signifie

$Y_0 = \text{nil}, \quad Y'_0 = \text{nil}$

et pour tout $n \geq 1$, $Y_n = X_{n-1}$ et $Y'_n = X'_{n-1}$

Exemple : détection de dépassement de seuil

```
distance = if (X>pre(X))
               then X-pre(X)
               else pre(X)-X ;
over = (distance > Threshold) ;
```



3.1. pre et ->

L'opérateur -> (suivi de)

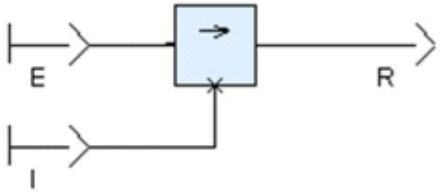
Permet d'initialiser un flot ou un ensemble de flots

Soit

X le flot (X_0, X_1, … , X_n, …) et Y le flot (Y_0, Y_1, … , Y_n, …)

alors

`R = Y -> X` est le flot (Y_0, X_1, … , X_n, …)



Par extension, l'équation

```
(Z, Z') = (Y, Y')->(X, X')
```

signifie

$Z_0 = Y_0$, $Z'_0 = Y'_0$ et pour tout $n \geq 1$, $Z_n = X_n$ et $Z'_n = X'_n$

Exemple : surveillance d'une température

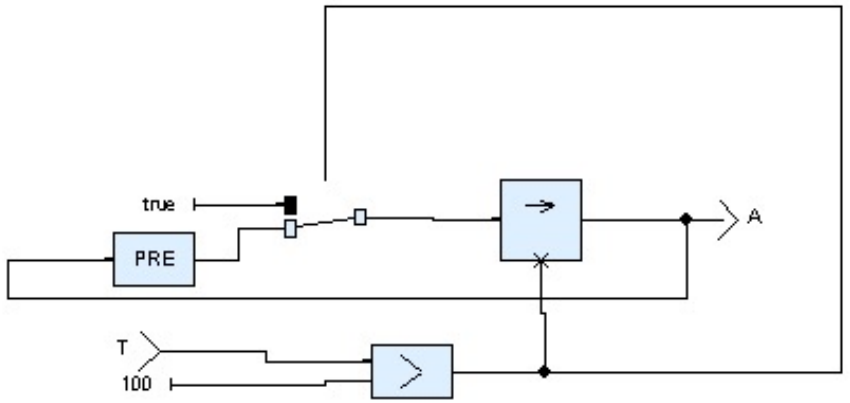
```
A = (T>100) ->
          if (T>100)
          then true
          else pre(A) ;
```

équivalent à :

$A_0 = (T_0 > 100)$

$A_n =$ true si $(T_n > 100)$

$\{ A_{n-1}$ sinon



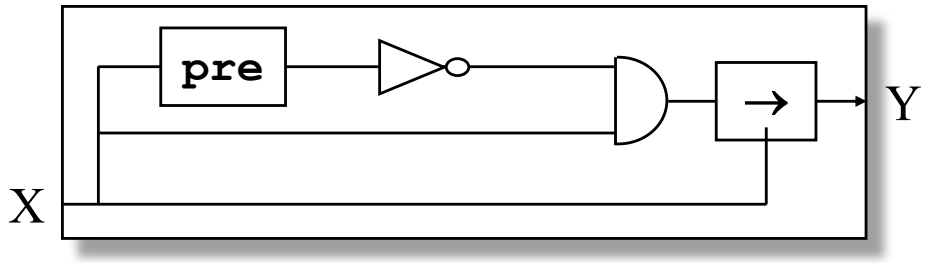
3.2. Exemples

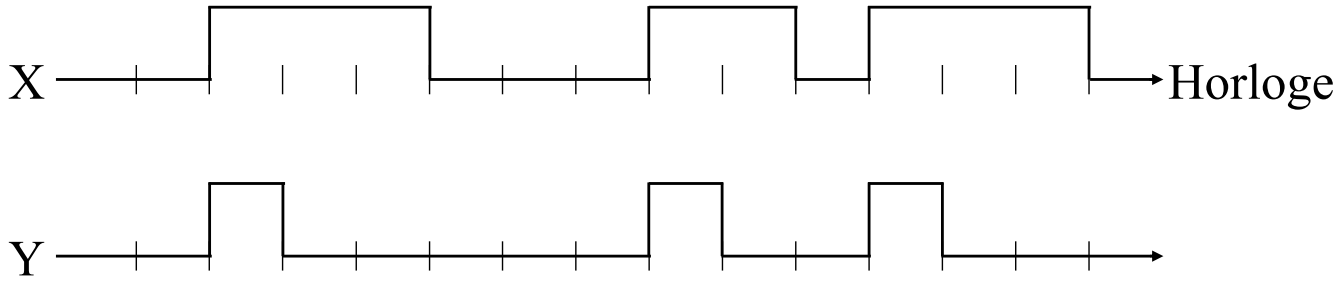
Exemple : détection de fronts montants

Soit X un flot d'entrée booléen

Soit Y un flot de sortie booléen

```
node rising_edge (X : bool) returns (Y : bool) ;
let
   Y = X -> (X and not pre(X));
tel ;
```



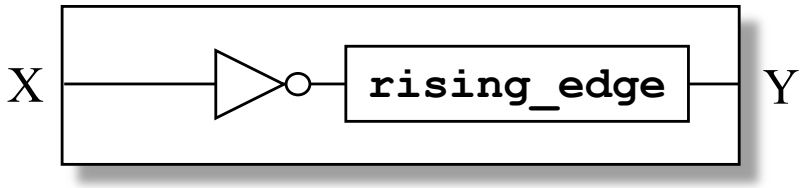


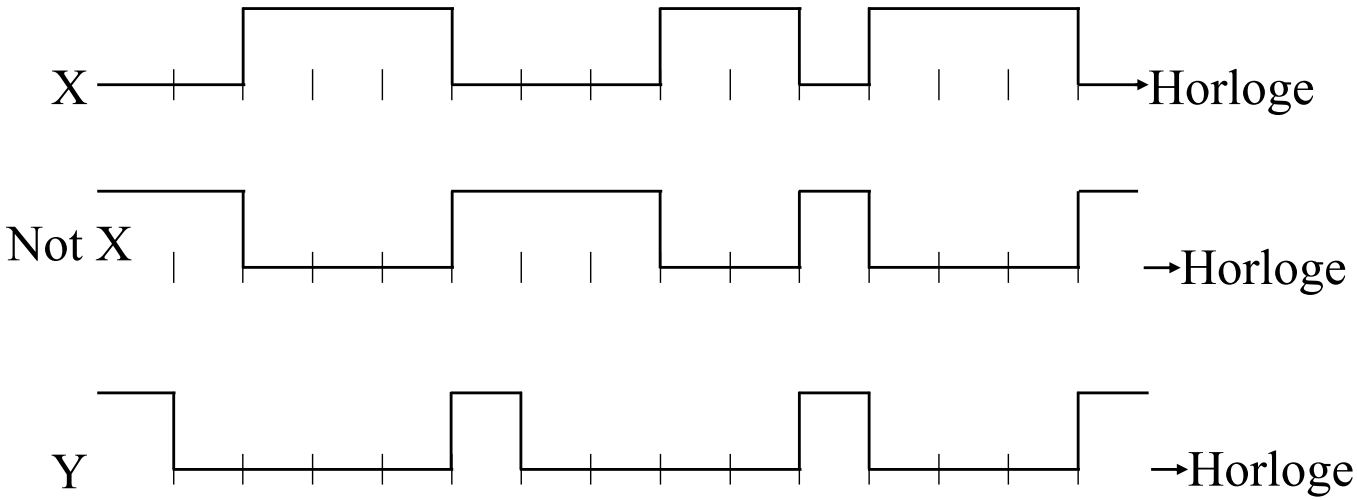
3.2. Exemples

Exemple : détection de fronts descendants

Réutilisation de l'opérateur **EDGE**

```
node falling_edge (X : bool) returns (Y : bool) ;
let
   Y = rising_edge (not X) ;
tel ;
```





3.3. Les assertions

La notion d'assertion

Permet au concepteur d'expliciter les hypothèses faites sur l'environnement et/ou sur le programme lui-même

=> permet d'optimiser la compilation

=> permet la vérification de propriétés sous conditions

=> simplifie la conception des programmes

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

3.3. Les assertions : exemple

Exemple : Justin, le loup, la chèvre et le chou…

Les entrées du programmes : les actions de Justin

- m : Justin traverse la rivière seul
- mw : Justin traverse la rivière avec le loup
- mg : Justin traverse la rivière avec la chèvre
- mc : Justin traverse la rivière avec le chou

=> flots booléens

b_n = vrai signifie que l'action b est effectuée à l'instant *n*

b_n = faux signifie que l'action b n'est pas effectuée à l'instant *n*

Hypothèse : les actions sont instantanées

Les sorties du programmes

- J : position de Justin
- W : position du loup
- G : position de la chèvre
- C : position du chou

=> flots entiers à valeurs dans {0, 1, 2}

X_n = 0 signifie X est sur la rive 0 à l'instant *n*

X_n = 1 signifie X est sur la rive 1 à l'instant *n*

X_n = 2 signifie X a été mangé à l'instant *n*

3.3. Les assertions : exemple

Exemple : Justin, le loup, la chèvre et le chou…

```
node justin(m, mw, mg, mc : bool) returns (J, W, G, C : int);
let
   assert (m or mw or mg or mc);
   assert( not (m and mw));
   assert( not (m and mg));
   assert( not (m and mc));
   assert( not (mw and mg));
   assert( not (mw and mc));
   assert( not (mg and mc));
   assert( true -> not (mw and not (pre(J)=pre(W))));
   assert( true -> not (mg and not (pre(J)=pre(G))));
   assert( true -> not (mc and not (pre(J)=pre(C))));

   J = 0 -> 1 - pre(J);
   W = 0 -> if mw then 1 - pre(W) else pre(W);
   G = 0 -> if pre(G) = 2 then pre(G)
            else if mg then 1 - pre(G)
                     else if (pre(G)=pre(W) and not mw) then 2
                              else pre(G);
   C = 0 -> if pre(C) = 2 then pre(C)
            else if mc then 1 - pre(C)
                     else if (pre(C)=pre(G) and not mg) then 2
                              else pre(C);
tel.
```

3.3. Les assertions : exemple

Exemple : Justin, le loup, la chèvre et le chou…

```
…
J = 0 -> 1 - pre(J);
W = 0 -> if mw then 1 - pre(W) else pre(W);
G = 0 -> if pre(G) = 2 then pre(G)
              else if mg then 1 - pre(G)
                       else if (pre(G)=pre(W) and not mw) then 2
                                else pre(G);
C = 0 -> if pre(C) = 2 then pre(C)
              else if mc then 1 - pre(C)
                       else if (pre(C)=pre(G) and not mg) then 2
                                else pre(C);
```

Séquence d'actions gagnante : . , mg, m, mw, mg, mc, m, mg

Que se passe-t-il pour la séquence : m, m, mw, m … ?

3.4. Les tableaux

Les tableaux

types scalaires

`bool`, `int`, `real`

=> types tableaux

`bool^4`, `int^n` avec n constante, `real^4^8`, ...

Mais : un programme Lustre doit s'exécuter en temps et espace borné. Les dimensions et indices des tableaux doivent donc être statiques et connus à la compilation

=> pré-compilation en Lustre sans tableau.

3.4. Les tableaux

Les tableaux : exemple

```
node Tdelay (const d : int; X : bool) returns(Y : bool);
var A : bool^(d+1);
let
   A[0] = X;
   A[1..d] = false^d -> pre(A[0..d-1]);
   Y = A[d];
tel


node Main (A : bool) returns (A_delayed : bool);
let
    A_delayed = Tdelay(10,A);
tel
```

3.5. La récursion

La récursion

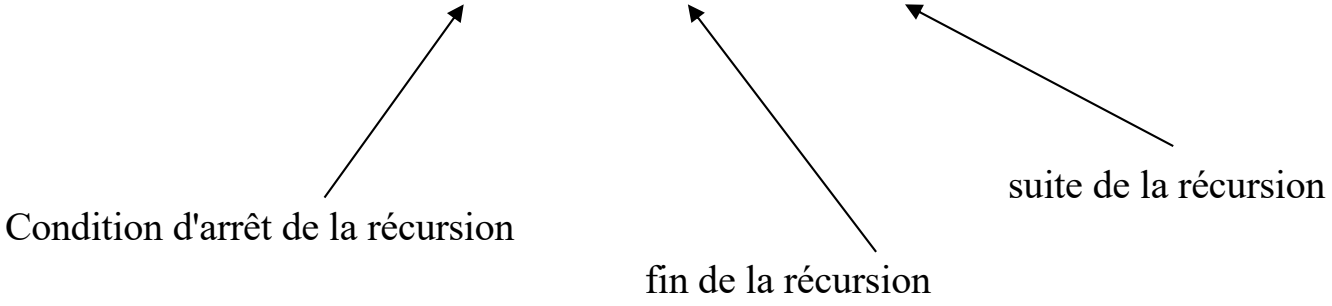
Un programme Lustre doit s'exécuter en temps et espace borné

=> la profondeur de la récursion doit être bornée statiquement et connue à la compilation

=> compilation en Lustre sans récursion

=> un opérateur conditionnel statique

```
X = with … then … else … ;
```



3.5. La récursion

La récursion : exemple

```
node Rdelay (const d : int; x : bool) returns(y : bool);
let
  Y = with d=0 then X else (false -> pre(Rdelay(d-1, X)));
tel


node Main (A : bool) returns (A_delayed : bool);
let
   A_delayed = Rdelay(10, A);
tel
```

4. La causalité

Problème de causalité :

Rappel : un réseau d'opérateurs de flots de données calcule un point fixe

Problème : un tel point fixe peut ne pas exister, être partiellement indéterminé, ou être multiple

=> programmes non causaux

Exemple :

`Y = X+Y`	non causal
`Y = X + pre(Y)`	indéterminé
`Y = X -> X + pre(Y)`	OK

=> Le rebouclage instantané des sorties sur les entrées est interdit

=> Le rebouclage retardé des sorties dur les entrées est autorisé à condition qu'il passe par au moins autant d'instance de"->" que de "pre"

Synchronous languages

Lecture 3: Lustre complet (basic + horloges)

Frédéric Boniol

ONERA - 2, av. E. Belin - 31055 Toulouse

frederic.boniol@onera.fr

Introduction

Première conclusion sur Lustre basic (sans horloge) :

Lustre basic est un langage permettant de décrire (naturellement) des systèmes cycliques :

- déterministes (les équations sont ordonnées de façon unique par l'ordre des flots)
- temps d'exécution borné (pas de processus dynamique, de boucle à longueur variables…)
- mémoire bornée (la profondeur des « pre » est bornée)
- modulaire (un nœud est un opérateur réutilisable sans effet de bord…)

Mais

En l'état, Lustre basic ne permet de concevoir que des systèmes "mono-cycles » (même « sampler » pour tout le monde)

=> extension « multi-cycle »

=> notion d'horloge

1. Lustre avec horloges

Notion d'horloge

Notion d'horloge

Une horloge est un flot booléen.

Horloge de base

On suppose qu'il existe une horloge de base, dénotée par le flot booléen toujours vrai (`true`).

=> L'horloge de base est le signal présent et vrai à chaque réaction du programme. C'est le signal qui caractérise les instants d'activation du programme.

=> Chaque nœud a une horloge de base locale, qui peut être l'horloge de base globale (l'horloge de base du nœud principal) ou une sous-horloge de celle-ci.

Horloge d'un flot

Chaque flot X est typé par une horloge (i.e., un flot booléen).

Un flot X est caractérisé par un couple (V,B) où :

V est la suite infinie ou finie de valeurs v_0, v_1, … v_n, …

B est l'horloge de X, i.e., une suite finie ou infinie de true et de false

=> X est présent avec la valeur v_n au nième instant ou B est vrai, et absent lorsque B est faux ou absent.

Notion d'horloge

Horloge et équation

Les équations doivent être homogènes du point de vue des horloges.

Exemple :

l'équation

```
Z = X + Y
```

n'a de sens que si X et Y ont la même horloge (et le même type), et définit un flot Z de même horloge (et de même type).

=> Toute équation `O=F(I,…)` définit un flot typé, c'est-à-dire :

- une suite de valeurs,
- une horloge, qui doit être construite de façon unique et non ambiguë.

=> Calcul des horloges.

Horloge d'un nœud

=> Un nœud peut recevoir des flots d'horloges différentes.

=> L'horloge d'un nœud (son horloge de base) est l'horloge de son entrée la plus rapide.

Lustre avec horloges

Opérateurs temporels

- opérateur de sous-échantillonnage sur une horloge moins rapide :

  `when`

- opérateur de sur-échantillonnage sur une horloge plus rapide :

  `current`

⇒ when et current sont les deux seuls opérateurs permettant de modifier l'horloge d'un flot.

- opérateur de sous + sur-échantillonnage

  `condact`

  ⇒Construite à partir de **`when`** et de **`current`**

Lustre avec horloges : when

Opérateurs de sous-échantillonnage : **when**

Projette un flot sur une horloge plus lente, permettant ainsi le dialogue d'un processus plus fréquent vers un processus moins fréquent.

Equivalent à un opération de « cast » (changement de type).

Soit le flot X et un flot booléen B (une horloge) de même horloge. L'équation

```
Y = X when B
```

définit un flot Y, de même type que X, et d'horloge B

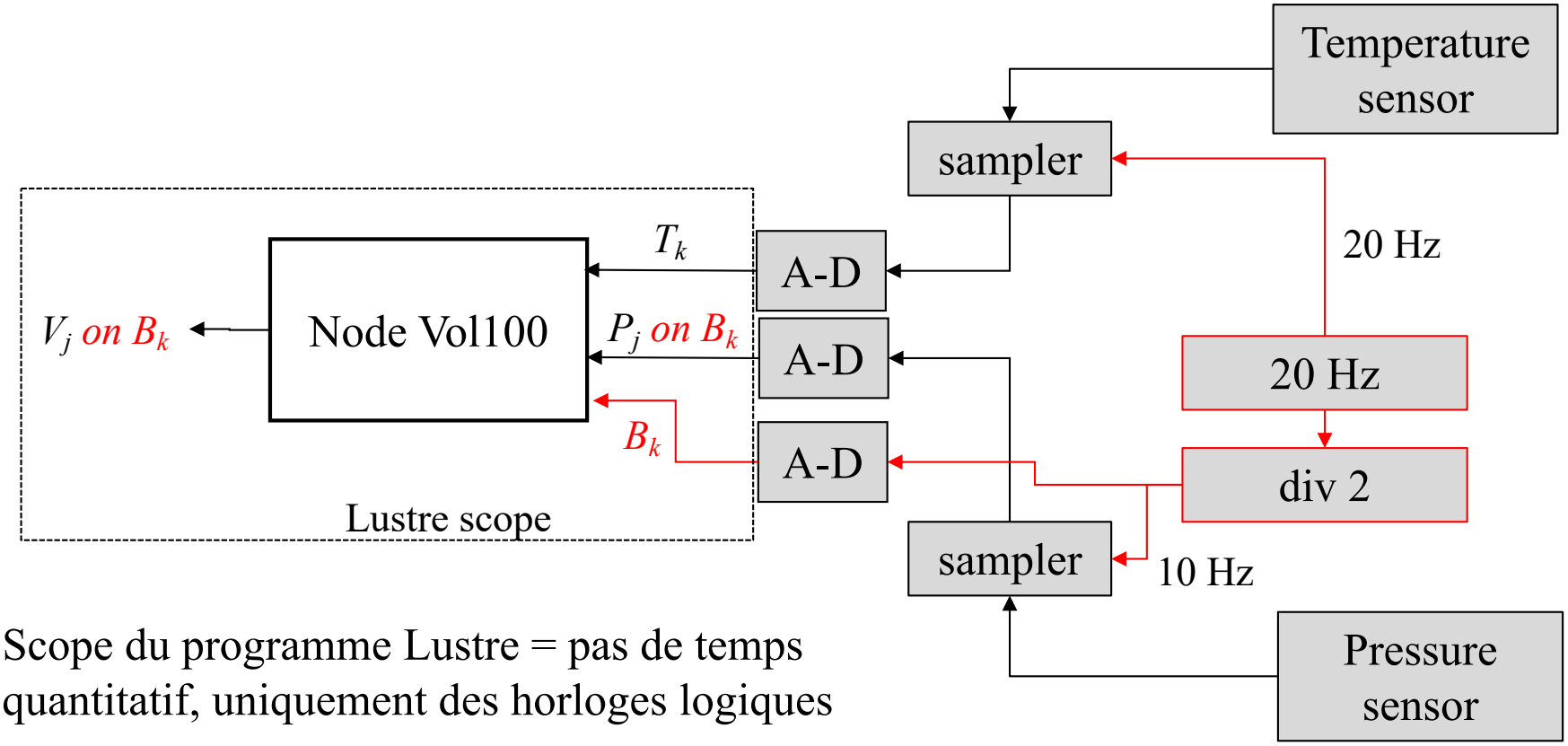
- Y est présent lorsque B est vrai
- Y est absent lorsque B est faux ou lorsque B et X sont absents

X	X0	X1	X2	X3	X4	X5	X6
B	**true**	false	false	**true**	**true**	false	**true**
Y = X when B	X0			X3	X4		X6

Lustre avec horloges : when

Opérateurs de sous-échantillonnage : `when`

Exemple : calcul d'un volume d'une mole de gaz tous les 100ms à partir d'une température et d'une pression arrivant respectivement tous les 50ms (horloge de base) et tous les 100ms :



⇒ Scope du programme Lustre = pas de temps quantitatif, uniquement des horloges logiques

Lustre avec horloges : when

Opérateurs de sous-échantillonnage : **when**

Exemple : calcul d'un volume d'une mole de gaz tous les 100ms à partir d'une température et d'une pression arrivant respectivement tous les 50ms (horloge de base) et tous les 100ms :

```
const R = 8.314;
node VOL100 (T:real; B:bool; P:real when B)
returns (V:real when B)
var T100:real when B;
assert (true -> (B or pre(B));
assert (true -> not (B and pre(B));
let
  V= (T100 / P) * (R when B);
  T100 = T when B;
tel.
```

Lustre avec horloges : current

Opérateurs de sur-échantillonnage : `current`

Projette un flot sur une horloge plus rapide, permettant ainsi le dialogue d'un processus moins fréquent vers un processus plus fréquent.

Soit le flot X et un flot booléen B (une horloge) de même horloge. L'équation

`Y = current X`

définit un flot Y, de même type que X, et d'horloge l'horloge de B

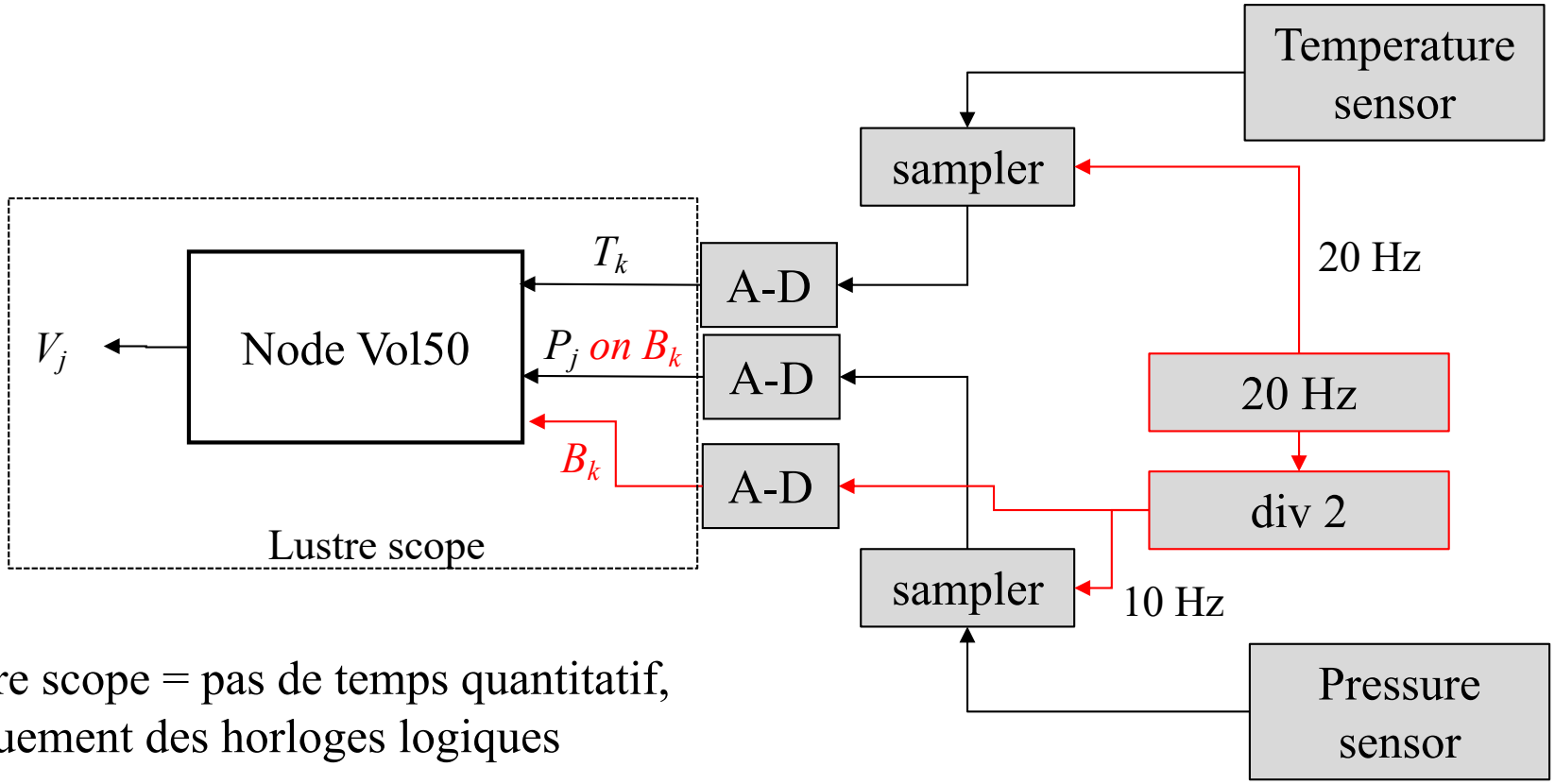
- Y est présent si et seulement si B est présent
- lorsque Y est présent, Y est égal à X si X est présent, sinon à la dernière valeur de X

X	X0	X1	X2	X3	X4	X5	X6
B1	**true**	false	false	**true**	**true**	false	**true**
B2	**true**	**true**	false	false	**true**	**true**	false
X1 = X when B1	X0			X3	X4		X6
B21 = B2 when B1	**true**			false	**true**		false
X21 = X1 when B21	X0				X4		
current (X1)	X0	X0	X0	X3	X4	X4	X6
current (X21)	X0			X0	X4		X4

Lustre avec horloges : current

Opérateurs de sur-échantillonnage : `current`

Exemple : calcul d'un volume d'une mole de gaz tous les 50ms à partir d'une température et d'une pression arrivant respectivement tous les 50ms (horloge de bas) et tous les 100ms :



⇒ Lustre scope = pas de temps quantitatif, uniquement des horloges logiques

Lustre avec horloges : current

Opérateurs de sur-échantillonnage : **current**

Exemple : calcul d'un volume d'une mole de gaz tous les 50ms à partir d'une température et d'une pression arrivant respectivement tous les 50ms (horloge de bas) et tous les 100ms :

```
const R = 8.314;
node VOL50 (T:real; B:bool; P:real when B)
returns (V:real)
var P50:real;
assert (true -> (B or pre(B));
assert (true -> not (B and pre(B));
let
  V= (T / P50) * R;
  P50 = current P;
tel.
```

Lustre avec horloges : exemple

Exemple : une minuterie simple (sur les ticks de base)

Entrée : set	activation de la minuterie (flot booléen)
sortie : level	état de la minuterie (flot booléen)
constante : delay	durée de la minuterie en top de l'horloge de base

```
node stable (set : bool; delay : int)
returns (level : bool)
var count : int;
let
   level = (count > 0);
   count = if set then delay
           else  if (false -> pre(level))
                 then pre(count) - 1
                 else (0 -> pre(count));
tel.
```

Lustre avec horloges : exemple

Exemple : une minuterie paramétrée par le temps

Entrée :	set	activation de la minuterie (flot booléen)
	second	unité de temps de la minuterie (flot booléen)
sortie :	level	état de la minuterie (flot booléen)
constante :	delay	durée de la minuterie en secondes

Idée : réutiliser le nœud `stable`

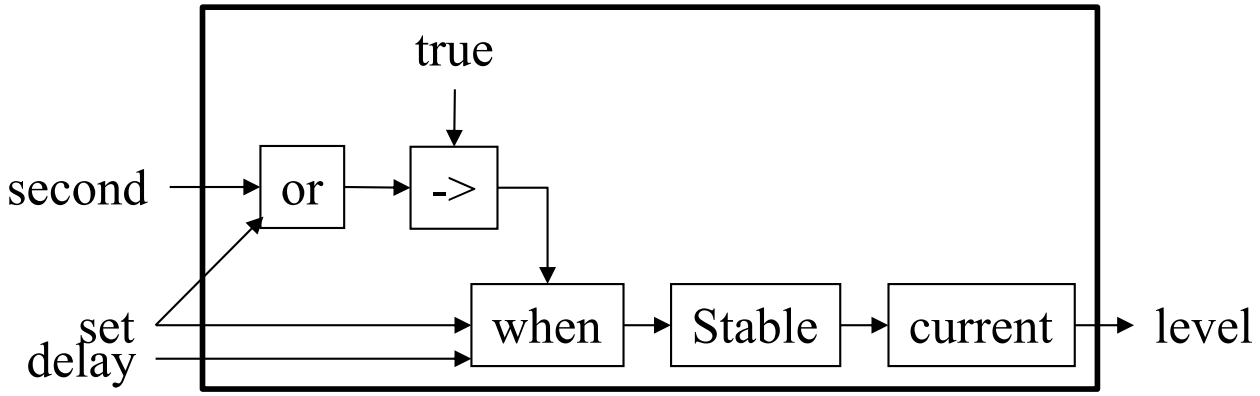
```
node stable_param (set, second : bool; delay : int)
returns (level : bool)
var ck : bool;
let
  level = current(stable((set, delay) when ck));
  ck = true -> (set or second);
tel.
```

=> Le nœud `stable` est appelé chaque fois que `ck` est vrai

=> L'horloge de l'instance de `stable` est `ck`

Lustre avec horloges : exemple

```
node stable_param (set, second : bool)
returns (level : bool)
var ck : bool;
let
  level = current(stable(set when ck));
  ck = true -> (set or second);
tel.
```



set	F	T	F	F	F	F	F	F	F	T
second	F	T	T	F	T	T	T	F	T	F
ck	T	T	T	F	T	T	T	F	T	T
count	0	5	4		3	2	1		0	5
level	F	T	T	T	T	T	T	T	F	T

Lustre avec horloges : à quoi ça sert ?

A contrôler les instants où un nœud est exécuté

Exemples

- Nœuds périodiques
- Nœuds soumis à des préconditions
- Nœuds exclusifs…

=> Voir BE2

Lustre : synthèse

Vérification sémantique :

Un programme Lustre est triplement vérifié

- vérification de types
- vérification de la causalité
  - Une et une seule équation par flot interne ou de sortie
  - Pas d'équation pour les flots d'entrée
  - Les dépendances de données (instantanées) entre équations forment un graphe acyclique
- **vérification d'horloge (vérification que le programme est bien « synchronisé »)**

⇒ Un programme Lustre qui compile est exécutable !

⇒ Et possibilité de vérifier formellement (parfois) le comportement d'un programme

2. Règles informelles du calcul d'horloge

Rappel syntaxique

Rappel syntaxique : un programme LUSTRE est composé

- d'une partie déclarative

```
X : type;
X : type when B ;
```

Soit *input*, *local*, *output* les listes des déclarations des flots d'entrée, locaux, de sortie.

- d'une partie équationnelle

```
Y = exp ;
```

```
exp ::= f(exp1,…, expn)        (avec f = +, -, *, /, or…)
      | exp1 when exp2
      | current(exp)
      | pre(exp)
      | exp1 -> exp2
      | X                      (flot)
      | val                    (valeur littérale ou constante)
```

Principes généraux du calcul d'horloge

Calcul d'horloge =

1. On calcule les fonctions suivantes

   - *clk_dec* : flot -> exp booléenne ∪ {`all`}

   fonction qui associe à tout flot son « horloge déclarée »

   - *clk_inf* : exp -> exp booléenne ∪ {`all`}

   fonction qui associe à toute expression de flot son « horloge inférée »

2. Puis on vérifie que *clk_dec* et *clk_inf* sont égales pour tous les flots du programme.

Règles informelle du calcul d'horloge

Horloge déclarée :

1. Tous les flots déclarés sans **when** ont comme horloge déclarée l'horloge de base du noeud

2. Tous les flots déclarés avec **when** ont comme horloge déclarée l'horloge déclarée par le **when**

Example:

```
node VOL100 (T:real; B:bool; P:real when B)
returns (V:real when B)
var T100:real when B;
```

clk_dec(T)= true

clk_dec(B)= true

clk_dec(P)= B

clk_dec(V)= B

clk_dec(T100)= B

Règles informelle du calcul d'horloge

Horloge inférée :

1. L'horloge inférée pour les flots d'entrée est leur horloge déclarée

2. Les constantes sont sur l'horloge de base

3. Soit une expression `f(exp1,…, expn)` avec f = `pre, ->, +, -, *, /, or`… (sans **`when`** ni **`current`**), alors
   - On ne peut inférer l'horloge de l'expression que si tous les exp_i ont la même horloge inférée
   - Et si c'est le cas, l'horloge inférée de l'expression est l'horloge inférée de chaque exp_i

4. Soit l'expression `X` **`when`** `B`, alors
   - On ne peut inférer l'horloge de l'expression que si X et B ont la même horloge inférée
   - Et si c'est le cas, l'horloge inférée de `X` **`when`** `B` est B

5. Soit l'expression **`current`** `X`, alors
   - On ne peut inférer l'horloge de l'expression que si l'horloge inférée de X n'est pas **`true`**
   - Et si c'est le cas, l'horloge inférée de **`current`** `X` est l'horloge inférée de l'horloge inférée de X

Règles informelle du calcul d'horloge

Horloge inférée :

Example:

```
const R : real;
node VOL50 (T:real; B:bool; P:real when B)
returns (V:real when B)
var T100:real when B;
let
  V= (T100 / P) * (R when B);
  T100 = T when B;
tel.
```

clk_dec(T)= true
clk_dec(B)= true
clk_dec(P)= B
clk_dec(V)= B
clk_dec(T100)= B

=>

clk_inf(T)= true
clk_inf(B)= true
clk_inf(P)= B

=>

clk_inf(T100)= B
clk_inf(P)= B

clk_inf(B)= B

Règles informelle du calcul d'horloge

Un programme est bien « synchronisé » (i.e., correct du point de vue des horloges) si :

- on est capable d'inférer une horloge pour tous les flots
- pour chaque flot, l'horloge déclarée et égale à l'horloge inférée

Synchronous languages

Lecture 4: vérification de programmes par model checking

Frédéric Boniol

ONERA - 2, av. E. Belin - 31055 Toulouse

frederic.boniol@onera.fr

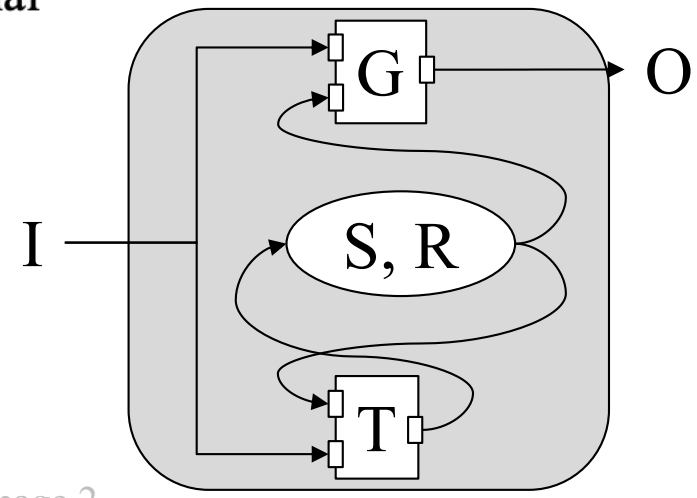
1. Principes (1/2)

Rappel : un programme Lustre décrit un comportement exprimable par un automate fini

⇒ représentable par une machine de Mealy déterministe.

Machine de Mealy déterministe : M = < S, I, O, T, G, s_0 >

- S = $\{s_0, s_1, \ldots, s_n\}$ ensemble fini d'états (état de contrôle)
- R = $\{r_0, r_1, \ldots, r_n\}$ ensemble fini de registres
- I = $\{i_0, i_1, \ldots, i_p\}$ ensemble fini de signaux d'entrée
- O = $\{o_0, o_1, \ldots, o_q\}$ ensemble fini de signaux de sortie
- T = fonction (totale) de transition : $S \times R \times 2^I \rightarrow S$
- G = fonction (totale) de génération des sorties : $S \times R \times 2^I \rightarrow O$
- s_0 = état initial

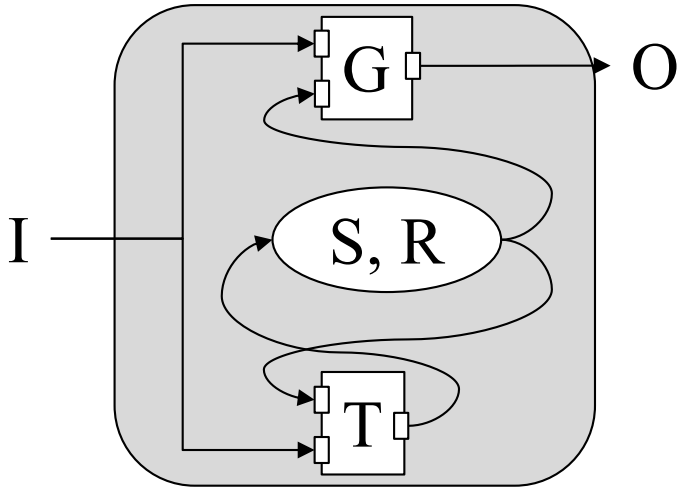


$$O_t = G((S,R)_t, I_t)$$
$$(S,R)_{t+1} = T((S,R)_t, I_{t+1})$$

1. Principes (1/2)

Lustre versus machine de Mealy déterministe :

- Notion d'état = valeur des variables bool
- Notion de registres = valeur des flots int et float
- Notion de transition (effet d'une réaction à des entrées sur les variables d'état et les sorties du programme).



$$O_t = G((S,R)_t, I_t)$$
$$(S,R)_{t+1} = T((S,R)_t, I_{t+1})$$

- Complétude / causalité des équations Lustre => G et T sont des fonctions totales (d'où le déterminisme)

1. Principes (1/2)

Example :

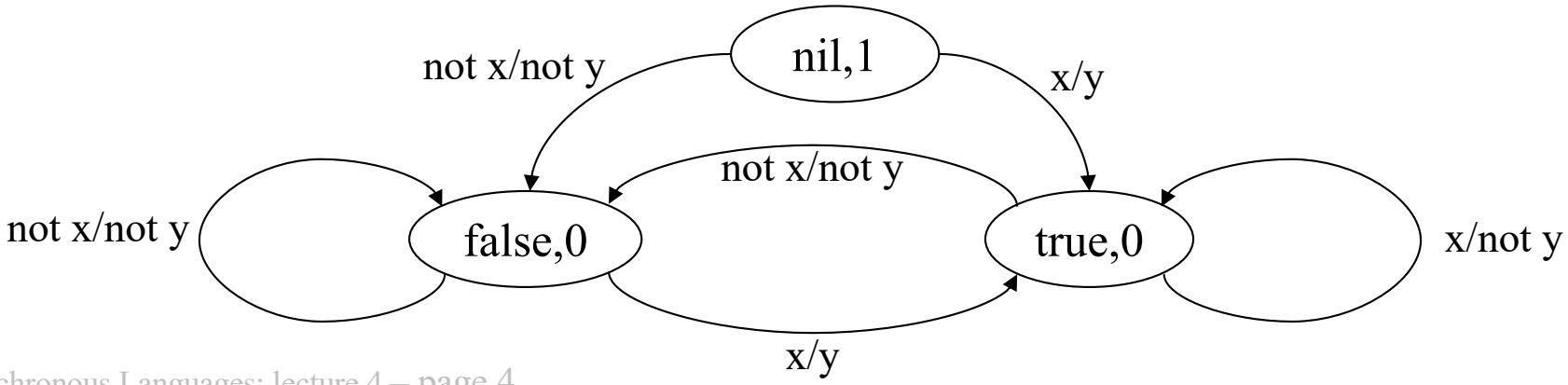
```
node rising_edge (X : bool) returns (Y : bool) ;
let
   Y = X -> (X and not pre(X));
tel ;
```

=> Fonction de réaction :

```
init = 1;
At each step :
        Y = if (init) X else (X && ! pX);
        pX = X
        init = 0
```

=> Etat = (pX, init), input = X, output = Y

=> Machine de Mealy sous-jacente



1. Principes (1/2)

Idée :

- En calculant les successeurs de chaque état atteint, on obtient l'arbre d'exécution infini du programme.
- En regroupant les états identiques (ayant les mêmes valeurs pour toutes les variables qui les composent), on obtient un graphe orienté représentant l'automate explicite du programme, déplié sur toutes les variables internes du programme (la machine de Mealy).
- On peut analyser les comportements du programme en étudiant la structure de ce graphe :
  - les états qui le composent (et les valeurs de certains signaux dans ces états),
  - les successions d'états le long d'un chemin,
  - les composantes fortement connexes, ...

  $\Rightarrow$ On peut « vérifier » que le programme satisfait des propriétés données

Question : quelles propriétés et comment les exprimer ?

1. Principes (2/2)

Idée :

- Lustre peut être vu comme une logique temporelle du passé permettant la spécification de comportements attendus… puis leur vérification

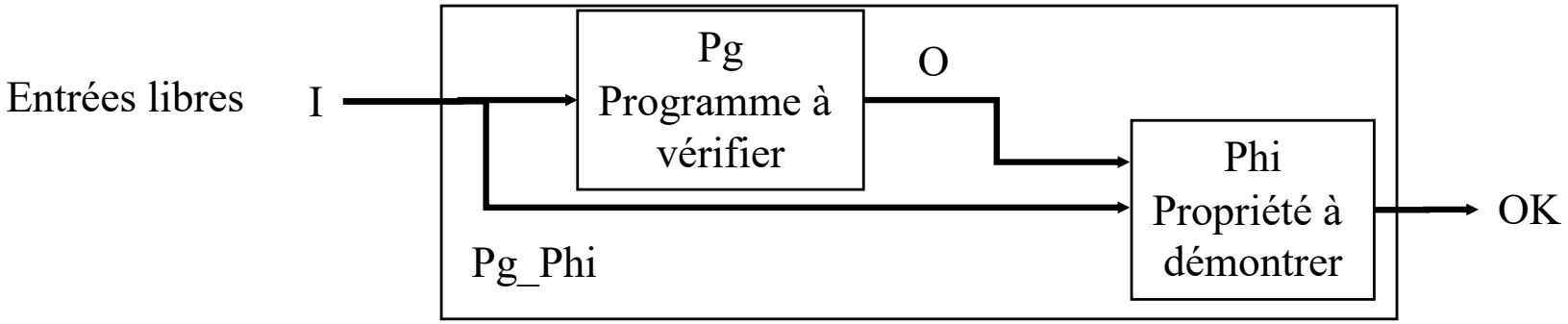
Mais

- Seuls les comportements attendus équivalents à des propriétés de sûreté peuvent être vérifiés (« une situation indésirable ne peut **jamais** se produire »)

Principe :

- expression des propriétés attendues sous la forme d'un nœud LUSTRE retournant un flot booléen OK (ce flot booléen dénote la valeur de vérité de la propriété attendue)
- composition de ce nœud avec le programme à vérifier et internalisation de toutes les sorties sauf OK
- L'ensemble $O = \{OK\}$ est l'ensemble des sorties de la machine de Mealy correspondante
- Parcours de la machine de Mealy
  - S'il existe une séquence de transitions telle que OK=fasle, alors la propriété est déclarée fausse
  - Sinon, la propriété est déclarée vraie

2. Architecture de vérification



- Pg = programme à vérifier écrit en LUSTRE
- Phi = observateur (propriété à démontrer écrite en LUSTRE)

⇒ Pg_Phi = Pg || Phi = composition synchrone du programme et la propriété

⇒ Pg_Phi est un programme écrit en LUSTRE !

⇒ Phi est satisfaite ssi tous les états de Pg_Phi sont étiquetés par la valeur true sur le flot OK

⇒ Vérification de Pg = composition synchrone des machines de Mealy de Pg_Phi et de Phi et test des valeurs prises par OK dans chaque état
= model checking !

⇒ Outils : LESAR et NP-tools (intégrés dans SCADE)
permet la vérification de programmes mettant en œuvre des flots booléens et des flots numériques définis par des relations linéaires

2. Architecture de vérification

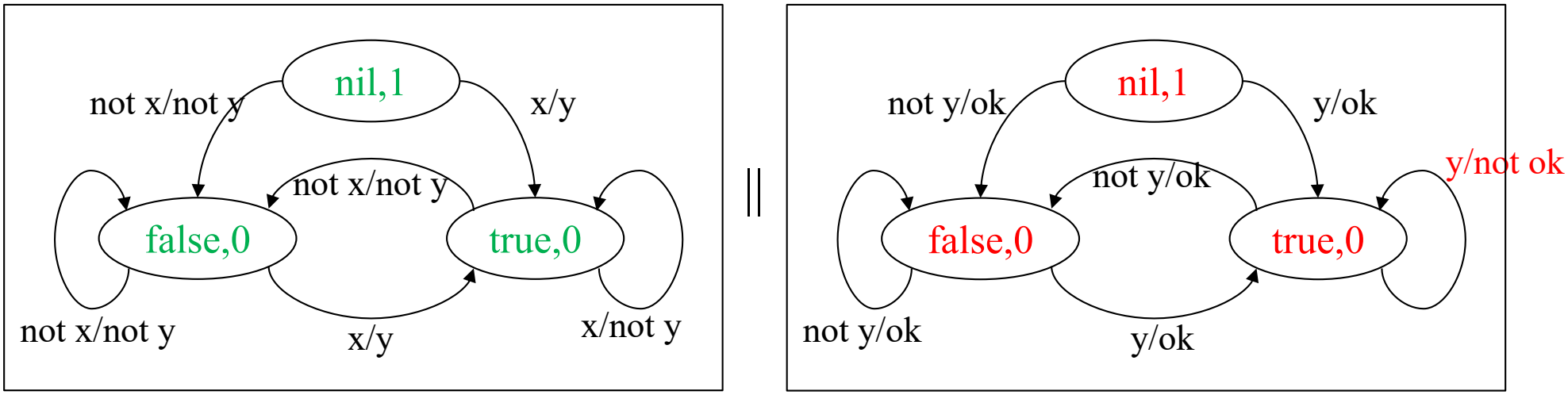
Example :

```
node rising_edge (X : bool) returns (Y : bool) ;
let
   Y = X -> (X and not pre(X));
tel ;
```

=> Propriété : Y n'est jamais vrai deux instants de suite

```
node phi (Y : bool) returns (OK : bool) ;
let
   OK = true -> (not (Y and pre(Y));
tel ;
```

=> Machines de Mealy sous-jacente



rising_edge phi

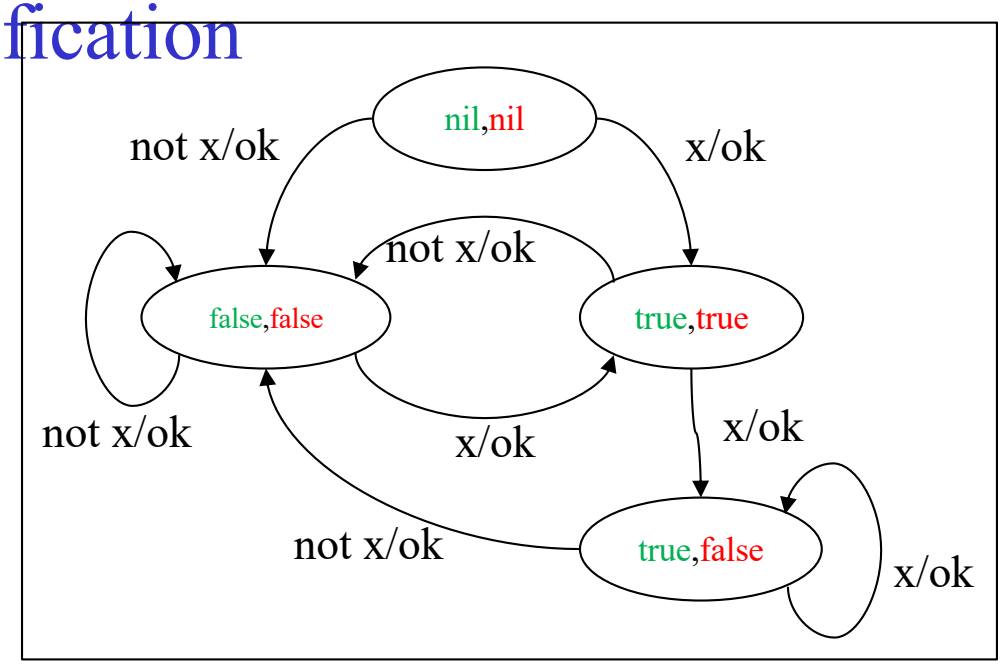
2. Architecture de vérification

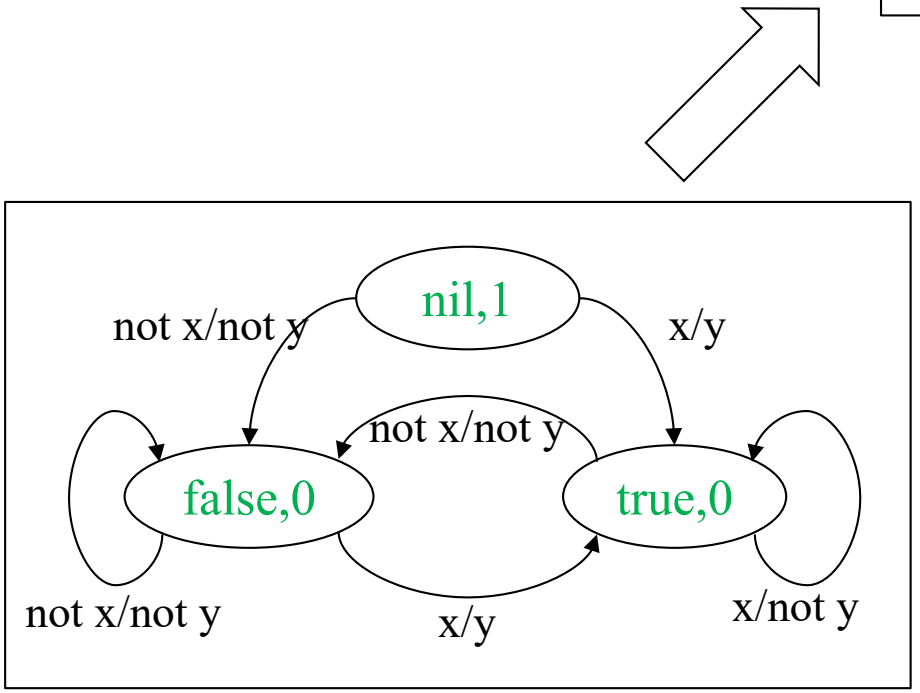
Example :

=> Composition synchrone

=> Ok est toujours vrai

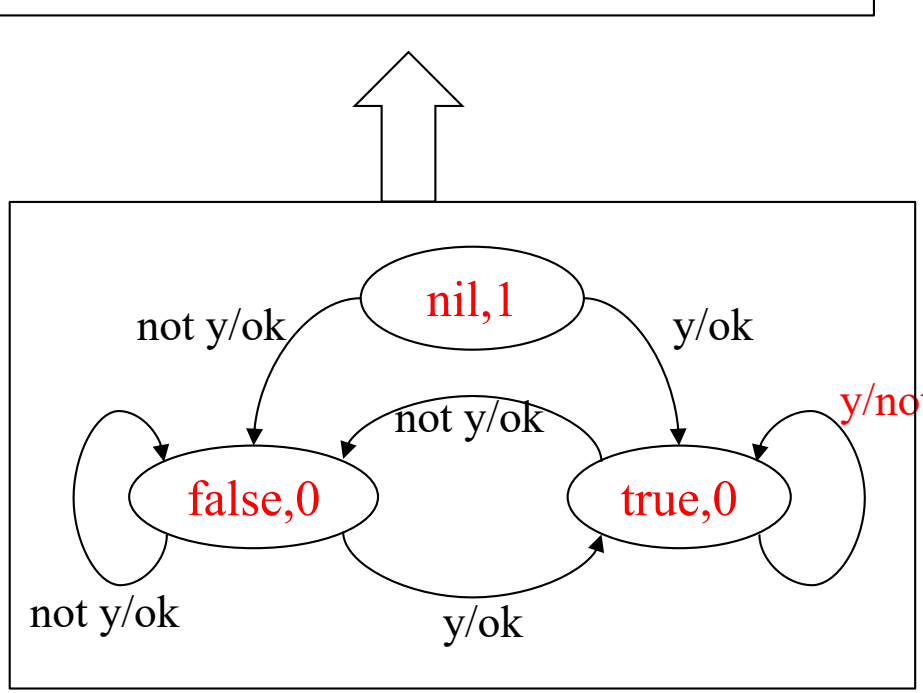
=> La propriété est vraie !





rising_edge

||



phi

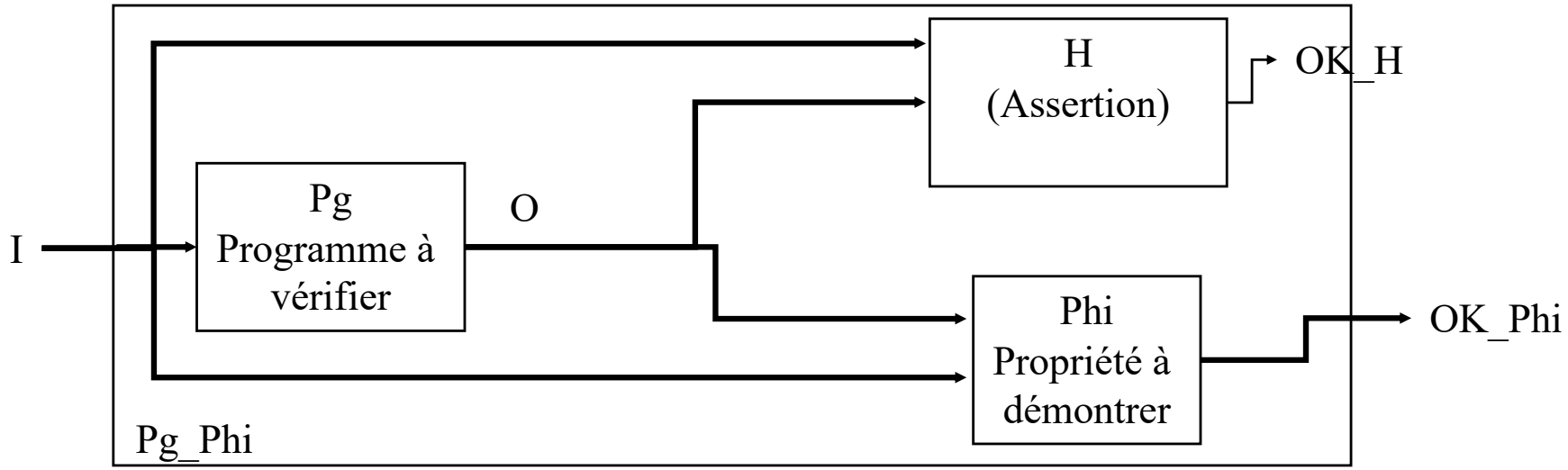
2. Architecture de vérification + assertions

Idée : Des assertions permettent de tenir compte de l'environnement du programme en exprimant des contraintes sur ses entrées en fonction de ses sorties

Une assertion est une formule logique sur les flots d'entrée et de sortie, qui doit être toujours vraie

⇒ Permet de vérifier que Pg satisfait Phi sous l'hypothèse que ses entrées et ses sorties satisfont l'assertion H

⇒ Permet de restreindre l'espace d'exploration



3. Expression des propriétés en Lustre

Observateur :

programme qui calcule un (et un seul) flot booléen en fonction de ses entrées (de type quelconque)

→ permet de modéliser des propriétés fonctionnelles attendues

Exemple :

```
node never (A : bool) returns (B : bool);
let
   B = not A -> (not A) and pre(B);
tel.
```

Never(X) est vrai tant que X est faux.

3. Expression des propriétés en Lustre

Exemple :

Soit la propriété attendue suivante

« toute occurrence de A doit être suivie par une occurrence de B avant la prochaine occurrence de C »

Peut être exprimée au passé par

« 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 »

Exprimable en LUSTRE par le nœud

```
node once_B_from_A_to_C (A, B, C: bool)returns (X: bool);
let
  X = C => (never(A) or since(B,A));
tel.

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

Synchronous languages

Lecture 5: récapitulatif et conclusion

Frédéric Boniol

ONERA - 2, av. E. Belin - 31055 Toulouse

frederic.boniol@onera.fr

1. Lustre : summary

- Node declaration: `node Name (input flows) returns (output flows);`
- Flow declaration:
  - for flow on the basic clock `F : type;`
  - for flow on a clock B `F : type when B;`
- Types: `int, real, bool`
- Flow definition (equation): `F = …;`
- Initialisation: `F -> …`
- Parallelism: `eq1 ; eq2 ;`
- Past operator: `pre(F)`
- Alternative operator: `if (…) then … else … ;`
- Sampling operator: `X when B`
- Oversampling operator: `current(F)`
- Node call: `MyNode(F)`
- Tuples: `(F1, F2) = (E1, E2);`
  `(F1, F2) = MyNode(E);`
- Boolean values: `true, false`
- Boolean operators: `or, and, nor, =>`
- Arithmetic operators: `+, -, *, /, div, mod, **, <>, =,`
  `<, >, <=, >=, (int)(F), (real)(F)`

1. Lustre : summary

Behaviour

B	false	true	false	true	false	false	true	true
X	x0	x1	x2	x3	x4	x5	x6	x7
Y	y0	y1	y2	y3	y4	y5	y6	y7
pre(X)	nil	x0	x1	x2	x3	x4	x5	x6
Y->pre(X)	y0	x0	x1	x2	x3	x4	x5	x6
Z=X when B		x1		x3			x6	x7
T=current Z	nil	x1	x1	x3	x3	x3	x6	x7
pre(Z)		nil		x1			x3	x6
0->pre(Z)		0		x1			x3	x6

1. Lustre : summary

Synthèse

Lustre est un langage :

- simple, déterministe, exécutable,
- avec garantie que la mémoire nécessaire à l'exécution est bornée
- avec garantie que le temps d'exécution (exécution d'un cycle) est borné
- formel car reposant sur une sémantique mathématique
  - ⇒ modèle sous-jacent: machines de Mealy déterministe
  - ⇒ génération automatique de code (SACADE) qualifiée (suppression des tests sur la cible)
  - ⇒ possibilité de vérification formelle
- industriel
  - ⇒ diffusé par Esterel Technologie : environnement SCADE
  - ⇒ utilisé par Airbus, Dassault Aviation, Continental, Safran, Thales…

Mais : réservé aux domaines où on peut appliquer l'hypothèse synchrone

- ⇒ limité à la programmation de logiciels cycliques

1. About development cycle and code generation

Written by the design team

prog.lus

```
node foo(X…)
returns (Y…)…
```

Automatic code generation

lus2c

foo_ext.h

```
// imported definitions
#define delay 10

// imported nodes
void N(…) {
…}
```

Included in

```
void foo_reset(foo_ctx* ctx) {
…}
void foo_new_ctx(void* cdata){
…}
void foo_step(foo_ctx* ctx){
…}
void foo_I_X(foo_ctx* ctx, typeX val){
…}
void foo_O_Y(void* cdata, typeY);
```

foo.c + foo.h

Included in

```
// Output functions
void foo_O_Y(void* cdata, typeY){…}
// Reactive loop
void main {
   init
   ctx=foo_new_ctx(cdata)
   while (1) {
      val = input acquisition;
      foo_I_X(foo_ctx* ctx, typeX val);
      foo_step(ctx);
      outputs…
      wait for the next period;
}
```

robot.c

Integrated and compiled on

gcc robot.c foo.c

Written by the integration team