

# Méthodes formelles de développement (6/9)

Hubert Garavel  
Frédéric Lang  
Pascal Raymond  
Wendelin Serwe

# Session 6

## Programmation Sychrone

## But de cette session

- Présenter une méthode pour le développement de systèmes réactifs critiques.
- Le parallélisme est un concept de programmation, *pas forcément d'implantation*.
- Utilisé en vrai (Schneider, Airbus, Eurocopter etc).
- Aperçu de deux styles de langages synchrones (Lustre/Esterel).

## Partie I Principes

## Rappel : Systèmes embarqués réactifs

- Interaction permanente avec leur environnement  
≠ transformationnels (ex. compilateurs)
- Contraintes de temps de réponse  
≠ interactif (ex. IHM, browser etc)  
L'environnement est (en partie) le monde physique
- Ressources limitées (mémoire, puissance de calcul, énergie, etc.)

### Exemples :

- Contrôle/commande, transport (critiques)
- De plus en plus : partout (téléphonie, appareils ménagers, multimedia, etc.)

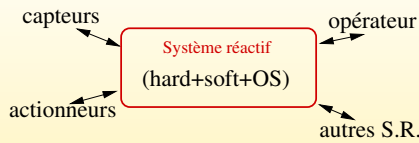
## Importance de l'informatique embarquée (?)

- Quel est le processeur le plus vendu (de tous les temps) ?  
« ARM franchit la barre des 10 milliards de processeurs »  
Ian Williams (Vnunet.com) 23-01-2008
- Extrait de l'article :  
« Les processeurs ARM équipent quasiment tous les types de produits électroniques, notamment téléphones, lecteurs multimédias, appareils photos numériques, télévisions HD, disques durs, périphériques et même systèmes de freinage automobile. »

### Remarques :

- à comparer avec ~ 2 milliards de PC vendus à ce jour
- ARM vend des licences, les vrais fabricants sont Intel, Texas Instruments, Samsung, STmicro, Siemens etc.

## Fonctionnement schématique

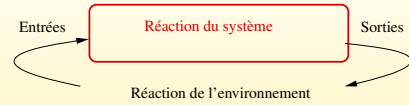


- Environnement : interface avec le monde physique, opérateur humain, autres systèmes réactifs...
- Le « programme » : un logiciel particulier, sur un système d'exploitation et une architecture particuliers ...

### Beaucoup de problèmes

⇒ On se concentre sur la *fonctionnalité du système*

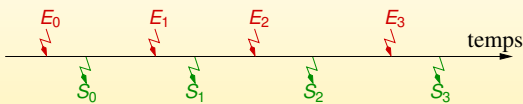
## Exécution d'un système réactif



- Entrées/sorties « informatiques » (booléens, entiers, flottants ...)
- Exécution = séquence de réactions



## Temps-réel ?



- les  $E$  et les  $S$  alternent dans le temps
- i.e., le programme répond à  $E_i$  **avant que** n'arrive  $E_{i+1}$
- i.e., le programme **ne rate aucun** changement significatif

## Temps-réel et fonctionnalité

- **Fonctionnalité** : le programme calcule les bonnes sorties  
Dépend essentiellement de la conception
- **Temps-réel** : le programme calcule assez vite  
Dépend aussi du matériel  
et de l'environnement considéré :
  - quelques dixièmes de seconde pour un humain  
(ex. 25 images/seconde = temps-réel pour un oeil humain)
  - quelques millièmes de seconde pour les processus physiques.
  - N.B. Les contraintes temps-réel sont fournies par un spécialiste du domaine.

## Fonctionnalité

Ici, on se concentre sur la fonctionnalité.

On veut garantir :

- **Déterminisme** :  
une séquence d'entrées donnée produit toujours la même séquence de sorties
- **Mémoire bornée** :  
on doit calculer les sorties avec une mémoire finie, allouée à l'avance  
(caractéristique essentielle des systèmes embarqués)

## Exemple

Un contrôleur de température :

- Entrées on, off (logique), temps (numérique).
- Sortie alarme vraie si allumé et temps > 30.

```
bool allume = false;
bool on, off, alarme;
float temps;
while(true) {
    lire(on, off, temps); // acquisition des entrées
    if (allume) {
        if (temps > 30.0) alarme = true; else alarme = false;
        if (off) allume = false;
    } else {
        if (on) allume = true;
    }
    écrire (alarme); // production des sorties
}
```

## Exemple (suite)

De manière plus abstraite :

- entrées (on, off, temps), sortie (alarme)
- une mémoire (allume), **rémanente** (utilisée d'une réaction sur l'autre)

A la fin de chaque réaction (ième boucle) :

- la sortie est **une fonction des entrées et de l'ancienne mémoire** :  
alarme = allume  $\wedge$  ( temps > 30.0 )
- la nouvelle mémoire est **une fonction des entrées et de l'ancienne mémoire** :  
allume' = si allume alors (non off) sinon (on)
- la mémoire est bien initialisée (ici allume = false à l'origine)

## Fonctionnalité d'un système réactif

La fonctionnalité de tout système réactif est (virtuellement) définie par :

- ses entrées/sorties ( $E, S$ )
- sa mémoire interne ( $M$ ), avec sa valeur initiale  $M_0$
- sa fonction de sortie :  
 $S_i = f(M_i, E_i)$
- sa fonction de transition :  
 $M_{i+1} = g(M_i, E_i)$

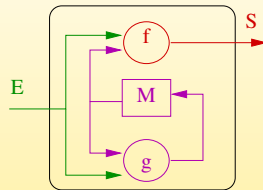
N.B. en général, les calculs de  $f$  et  $g$  sont mélangés dans une seule procédure (cf. l'exemple).

## Réalisation d'un système réactif simple

Implémentation « event-driven » :

```

Système (E, S)
mémoire M
M := M0
boucle
  attendre(E)
  S = f(M, E)
  M = g(M, E)
  écrire(S)
fin boucle
  
```



Temps-réel ?

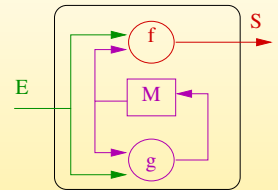
**temps de calcul < temps de réaction de l'environnement**

## Réalisation d'un système réactif simple (2)

Échantillonnage :

```

Système (E, S)
mémoire M
M := M0
à chaque période faire
  lire(E)
  S = f(M, E)
  M = g(M, E)
  écrire(S)
fin
  
```



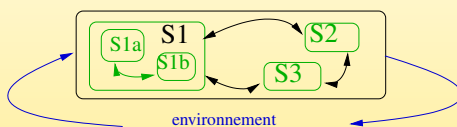
Temps-réel ?

**temps de calcul < période** et période connue pour l'environnement considéré.

## Système réactif complexe

Il faut découper le problème :

- gros système : beaucoup d'entrées/sorties
- conception monobloc impossible
- solution classique : conception parallèle et hiérarchique



Fonctionnement attendu : chaque sous-système se comporte localement comme un système temps-réel

## Parallélisme de description

- le parallélisme peut être imposé (système réparti),
- ou seulement logique (parallélisme de description) i.e., l'architecture est centralisée

On se place dans le dernier cas (mono-processeur)  
Comment implémenter un système complexe ?

## Implémentation par processus concurrents

- Un processus (ou thread) par sous-système.
- Communication/ordonnancement réalisés par l'exécutif :
  - primitives système (OS temps-réel)
  - primitives du langage lui-même (ex. ADA)

⇒ **Problème : fonctionnement global très difficile à prévoir :**

- temps de calcul difficile à cerner,
- ordre des communications difficile à maîtriser :
  - on peut influencer (priorités, communication bloquantes),
  - mais risque de bugs (blocage, inversion de priorité)

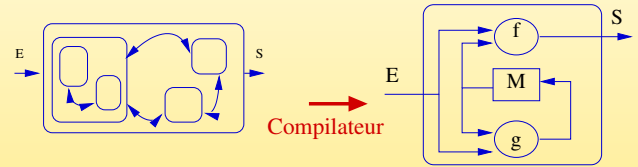
En bref :

- c'est très « artisanal »
- on risque de perdre le déterminisme !

## Approche synchrone

Concilier les avantages :

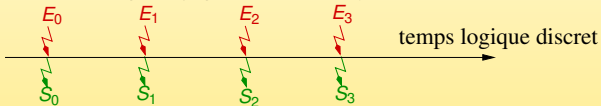
- d'une conception modulaire et parallèle
- du déterminisme et de l'aspect temps-réel de l'implémentation monobloc



## Hypothèse synchrone

Idéalement (conception) :

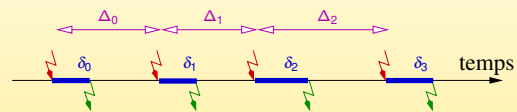
- Communication non bloquante en temps nul (diffusion synchrone)
- Réaction en temps nul
- Composition triviale :  $0 + 0 = 0$  (modularité idéale)
- En particulier, les sorties sont simultanées aux entrées
- D'où un temps logique discret (le rythme des entrées)



## Hypothèse synchrone

Concrètement (exécutable) :

- Réaction simple (pas d'itération, mémoire bornée),
- donc il existe une borne max au temps de calcul (WCET) évaluable pour une architecture donnée.



- soit  $\delta_{max}$  un majorant des  $\delta_i$  (pour l'implémentation considérée),
- soit  $\Delta_{min}$  un minorant des  $\Delta_i$  (cf. le spécialiste du domaine)
- l'hypothèse synchrone est valide si  $\delta_{max} < \Delta_{min}$

## Quoi de neuf dans tout ça ?

En fait, relativement classique :

- dans le domaine des circuits (conception en synchrone idéal, puis placement/routage adéquat pour respecter l'hypothèse)
- en automatique (Équations aux différences finies, réseaux analogiques, schémas à relais, ladder, grafcet)

Moins classique dans le domaine logiciel...

## Langages synchrones

Tous basés sur les même principes :

- conception avec un temps logique discret
- communication synchrone idéale
- génération de code simple et efficace

Mais plusieurs styles :

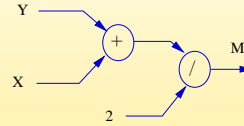
- Déclaratif, flot-de-données (Lustre)
- Impératif, séquentiel (Esterel)

## Partie II

### Le langage Lustre

## Approche flot de données

Classique en automatique et en conception de circuits



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

Interprétation synchrone : temps = IN

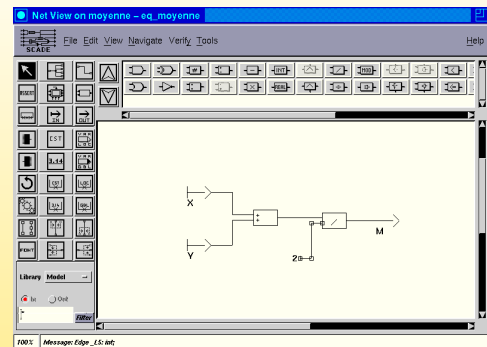
$$\forall t \in \mathbb{N} \ M_t = (X_t + Y_t) / 2$$

## Autre version

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

- une définition pour chaque sortie et variable auxiliaire
- ordre indifférent
- principe de substitution
- X, Y, M, S dénotent des séquences infinies de valeurs

## Lustre (académique) et SCADE (graphique)



## Lustre combinatoire

### Les types de base

- booléen (`bool`), entier (`int`), flottant (`real`)

### Les constantes

- $2 \equiv 2, 2, 2, 2, \dots$
- `true`  $\equiv$  *vrai, vrai, vrai, vrai, ...*

### Les opérateurs « point à point »

- opérateurs arithmétiques et logiques classiques

$$X \equiv X_0, X_1, X_2, X_3 \dots \quad Y \equiv Y_0, Y_1, Y_2, Y_3 \dots$$

$$\Rightarrow X + Y \equiv X_0 + Y_0, X_1 + Y_1, X_2 + Y_2, X_3 + Y_3 \dots$$

## Exemple booléen

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

Exécution :

X	vrai	vrai	faux	vrai	vrai	faux	...
Y	faux	vrai	faux	faux	vrai	faux	...
U	faux	vrai	faux	faux	vrai	faux	...
Z	vrai	faux	vrai	vrai	faux	vrai	...

## Exemple : l'opérateur if

```
node Max(A,B : real) returns (M : real);
let
  M = if (A >= B) then A else B;
```

```
tel
```

Erreur classique :

```
let
  if (A >= B) then M = A;
  else M = B;
tel
if : (flot bool) × (flot T) × (flot T) → (flot T)
```

## Mémoire

Opérateur « pre » :

- retard élémentaire

X	x <sub>0</sub>	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	x <sub>4</sub>	...
pre X	nil	x <sub>0</sub>	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	...

- i.e.  $(\text{pre}X)_0$  indéfini et  $\forall i \neq 0 (\text{pre}X)_i = X_{i-1}$

Opérateur « -> » :

- initialisation

X	x <sub>0</sub>	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	x <sub>4</sub>	...
Y	y <sub>0</sub>	y <sub>1</sub>	y <sub>2</sub>	y <sub>3</sub>	y <sub>4</sub>	...
X -> Y	x <sub>0</sub>	y <sub>1</sub>	y <sub>2</sub>	y <sub>3</sub>	y <sub>4</sub>	...

- i.e.  $(X \rightarrow Y)_0 = X_0$  et  $\forall i \neq 0 (X \rightarrow Y)_i = Y_i$

## Exemple : fronts montants

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

```
tel
```

X	faux	faux	vrai	vrai	faux	vrai	...
pre X	nil	faux	faux	vrai	vrai	faux	...
E	faux	faux	vrai	faux	faux	vrai	...

## Exemple : min et max d'une séquence

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

```
tel
```

X	12	5	7	-2	21	0	...
min	12	5	5	-2	-2	-2	...
max	12	12	12	12	21	21	...

⇒ Définition récursive de flot

## Définitions récursives correctes

- X peut dépendre de pre X
- ex. alt = false -> not pre alt
- ex. nat = 0 -> 1 + pre nat;

Les dépendances instantanées sont **interdites**

- le compilateur renvoie une erreur
- intuitivement : pas de court-circuit

## Modularité

- tout nœud défini par l'utilisateur peut être réutilisé
- ressemble à un « appel de procédure »

```
node MoyenneMinMax(X : int) returns (M : int);
var min, max : int;
let
  M = Moyenne(min, max);
  min, max = MinMax(X);
tel
```

On peut écrire directement « M = Moyenne(MinMax(X)) ; »

## Quelques exercices/exemples ...

- noeud « switch », deux entrées on, off, une sortie run
 

```
node Switch(on,off : bool) returns (run : bool);
let
  run = if (false -> pre run) then not off else on;
tel
```
- compter les occurrences de tic, avec un bouton reset
 

```
node Counter(tic,reset : bool) returns (cpt : int);
let
  cpt = if reset then 0
        else if tic then (0 -> pre cpt)
        else (0 -> pre cpt);
tel
```

## Partie III

### Le langage Esterel

## Principes de base

### Communication par signaux

- information élémentaire (intuition : signaux « radio »)
- peut être pur (juste absent ou bien présent) ...
- ... ou valué (absent ou bien présent avec une valeur)

### Comportements de base :

- relatifs aux signaux : émettre/attendre/lire un signal

### Plus des instructions « classiques » pour les composer :

- mise en séquence, en parallèle, en boucle
- test, interruptions

## Exemple : calcul de vitesse

Le langage est assez intuitif, voyons directement un exemple :

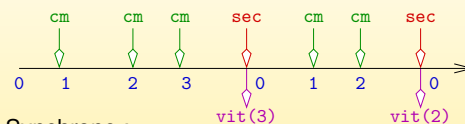
- un objet se déplace et reçoit deux signaux :
  - sec d'un horloge temps-réel
  - cm chaque fois que l'objet a parcouru 1 centimètre
- le programme doit envoyer, à chaque seconde, la vitesse en cm/s

## Exemple : calcul de vitesse (2)

```
module VITESSE :
input sec, cm;           % signaux purs
output vit : integer;   % signal valué
loop % comportement infini
  var cpt := 0 : integer in % variable interne
  abort % Termine le comportement suivant :
  loop % comportement normal :
    await cm;           % à chaque cm,
    cpt := cpt + 1 % incrémenter cpt
  end loop
  when sec do          % ... quand sec arrive,
    emit vit(cpt)      % et émet la vitesse avec la valeur cpt
  end abort
end var
end loop.
```

## Exemple : calcul de vitesse (3)

Comportement dans le temps :



Synchrone :

- sauf exception, les instructions sont instantanées,
- les exceptions sont :
  - await cm attendre la prochaine occurrence future de cm
  - abort ... when sec interrompre sur la prochaine occurrence future de sec

## Conclusion de l'exemple

- Langage impératif « assez classique »...
- ... mais avec une sémantique synchrone originale.

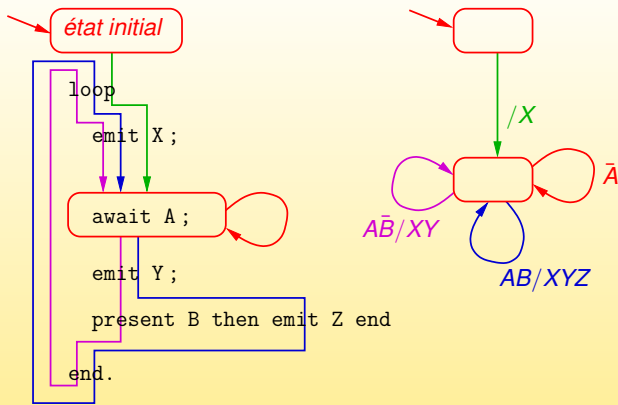
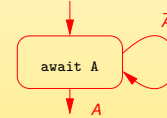
Pour mieux comprendre :

- on va se limiter aux signaux purs,
- on va voir qu'un programme Esterel est équivalent à un automate

## Esterel et automates

- Un programme Esterel est un automate
- Les points de contrôle (les états) sont :
  - le tout début du programme,
  - les instructions qui « prennent du temps » (i.e. les `await`)
- Les transitions sont étiquetées par un couple (condition/émission)

Le « bout » d'automate de base est le suivant :



## Opérateur parallèle

- `[ c1 || c2 ]`
- passse **immédiatement** le contrôle à `c1` **et** à `c2`,
- termine si et quand **le dernier des deux** termine.

Remarque : plusieurs branches parallèles peuvent émettre le même signal ...

- aucun problème pour les signaux purs,
- pour les signaux valués il faut définir une opération de combinaison (ex. addition)

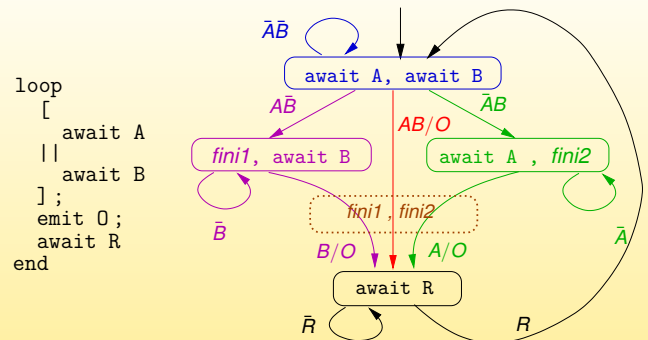
## Opérateur parallèle et automate

Produit d'automates assez classique, avec un traitement particulier pour la terminaison :

- on ajoute deux états « virtuels » `fini1` et `fini2`,
- les états sont des couples (*état de c1, état de c2*),
- l'état (`fini1, fini2`) est transitoire par définition (terminaison instantanée).

En bref, soient `await A` un état de `c1` et `await B` un état de `c2` :

- (`await A, await B`) est un état du produit,
- (`fini1, await B`) est un état du produit,
- (`await A, fini2`) est un état du produit,
- (`fini1, fini2`) n'est pas un état du produit !



## Partie IV

## Utilisation de langages synchrones

## Compilation

- Les compilateurs génèrent du code séquentiel simple dans un langage dit « hôte » (C en général)
- i.e. le parallélisme est « résolu » une fois pour toute (ordonnancement statique)
- Ils produisent uniquement la procédure “step” (un pas de calcul)
- La boucle infinie (le programme principal) est écrite à part, selon les besoins...

## Langage noyau vs langage hôte

- Lustre, Esterel sont volontairement limités
- Les données et les traitements complexes sont déclarés externes, et programmés directement dans le langage hôte
- ex. fonctions numériques en Lustre :
 

```
function sqrt(x : real) returns (s : real);
...
node toto (...) ...
  a = ... sqrt(b) ...
```
- le code produit par le compilateur est ensuite « lié » avec la librairie adéquate

## Validation formelle

- Les langages synchrones ont un sémantique formelle parfaitement définie.
- Les programmes sont des modèles formels :
  - évident pour Esterel (cf. Esterel et automates)
  - un peu moins direct, mais vrai aussi pour Lustre (un programme Lustre est un automate)

On peut donc appliquer des méthodes de vérification formelles sur les programmes (cf. model-checking).

## Utilisation dans l'industrie

## Scade (Lustre) :

- En opérationnel, dans les domaines critiques :
  - Schneider Electric (centrales nucléaires)
  - Airbus (commandes de vol A340-600, A380)
- En R&D un peu partout ...

## Esterel :

- Domaine des circuits (Texas-Instrument)
- En R&D un peu partout ...