

# Programmation Concurrente, Réactive et Répartie

## Cours N°5

Nadine Richard - Emmanuel Chailloux

Master d'Informatique  
Université Pierre et Marie Curie

année 2011-2012

# Plan du cours

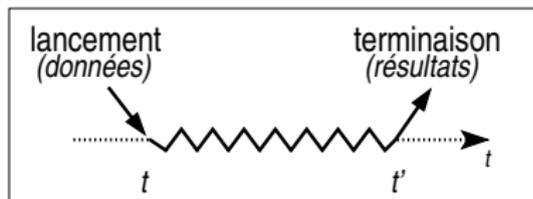
- ▶ Classification selon D. Harel et A. Pnueli
- ▶ Les systèmes réactifs
- ▶ Approche synchrone pour les systèmes réactifs
- ▶ Esterel : un langage synchrone impératif
- ▶ Machine d'exécution et compilation
- ▶ Conclusion
- ▶ Autres applications du modèle synchrone
- ▶ Références

# Classification

- ▶ Systèmes transformationnels
- ▶ Systèmes interactifs et réactifs

# Systèmes transformationnels

- ▶ Effectue des calculs :
  - ▶ à partir des données fournies
  - ▶ pour produire des résultats
  - ▶ puis se terminer.



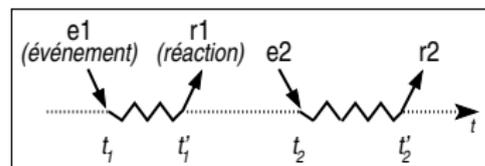
- ▶ Exemples : compilateur/traducteur, éditeur de factures

# Systèmes interactifs et réactifs

- ▶ Interagit continuellement avec son environnement :
  - ▶ doit fournir une réponse aux événements reçus
  - ▶ ne se termine pas !

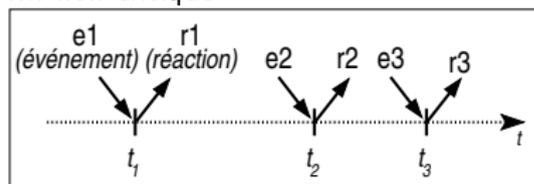
- ▶ Interactif

- ▶ réagit à son rythme
- ▶ exemples : base de données, IHM non critique



- ▶ Réactif

- ▶ réagit au rythme imposé par l'environnement
- ▶ exemples : contrôle de processus industriels, IHM critique



# Systèmes réactifs

- ▶ Théorie et pratique
- ▶ Approches traditionnelles
- ▶ Approche synchrone
- ▶ Systèmes réactifs temps-réel
- ▶ Langages et outils synchrones

# Systèmes réactifs : théorie et pratique

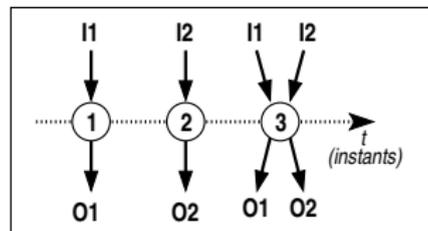
- ▶ En théorie : réaction en temps nul
- ▶ En pratique : implémentés par des systèmes interactifs suffisamment rapides
  - ▶ pour prendre en compte tous les stimuli
  - ▶ pour y répondre à temps

# Approches traditionnelles

- ▶ Automate
  - ▶ comportement déterministe
  - ▶ exécution efficace
  - ▶ mais : faible maintenabilité
  
- ▶ Langage + exécutif temps-réel multi-tâches
  - ▶ tâches coopérantes + communication asynchrone
  - ▶ primitives pour le parallélisme et la synchronisation (tâches et rendez-vous en Ada)
  - ▶ mais : déterminisme non garanti

# Approche synchrone

- ▶ Échelle de temps logique discret



- ▶ instant = réaction du système
  - ▶ réaction en temps nul : une réaction commencée doit se terminer avant que débute la suivante, donc avant qu'un nouvel événement arrive
  - ▶ signaux en entrée et en sortie simultanés
- ▶ Fondements mathématiques
  - ▶ déterminisme garanti
  - ▶ composition de systèmes synchrones  $\Rightarrow$  système synchrone
  - ▶ outils automatiques de vérification formelle

# Systèmes réactifs temps-réel

- ▶ Système temps-réel
  - ▶ résultat correct + respect des contraintes temporelles  
⇒ résultat faux s'il arrive trop tard !
  - ▶ contraintes temporelles : strictes ou souples (délai moyen)
  - ▶ systèmes critiques : *safety critical* ou *mission critical*
- ▶ Caractéristiques principales
  - ▶ prévisibilité = parfaitement déterministe
  - ▶ sûreté = comportement garanti (situations extrêmes)
- ▶ ... d'où l'utilisation de systèmes réactifs synchrones !

# Langages et outils synchrones

- ▶ Langages impératifs (signaux discrets)  
Esterel (*INRIA, ENMP, CMA*), Marvin (*ENST*)
- ▶ Langages à «flots de données» (signaux continus)  
Signal (*INRIA*), Lustre (*IMAG*), Lucid Sychrone (*INRIA, LIP6, IMAG*)
- ▶ Formalismes (description graphique d'automates)  
SyncCharts (proche d'Esterel), Argos (proche de Lustre), HPTS (proche de Signal)
- ▶ Esterel Technologies : Esterel studio  
compilateur Esterel, éditeur de SyncCharts, simulateur interactif, ...

# Le langage Esterel

- ▶ Caractéristiques et principes du langage
- ▶ Cycle de vie d'une instruction
- ▶ Modules
- ▶ Expressions de signaux, émission instantanée
- ▶ Immédiat vs. différé
- ▶ Manipulation de données
- ▶ Types, fonctions et procédures externes
- ▶ Prémption et trappes
- ▶ Instructions dérivées
- ▶ Composition de modules
- ▶ Tâches asynchrones

# Caractéristiques du langage

- ▶ Langage réactif synchrone de nature impérative
  - ▶ instructions impératives : séquence, composition parallèle, ...
  - ▶ instructions réactives : pause, attente d'événement, ...
  
- ▶ Avec :
  - ▶ programmation modulaire
  - ▶ manipulation de données : types prédéfinis ou externes
  - ▶ mécanismes de trappe et de préemption
  - ▶ gestion des tâches asynchrones

# Principes fondamentaux

## ▶ Instant

- ▶ instant  $t$  de début d'une instruction (1<sup>er</sup> instant)
- ▶ instant  $t'$  de terminaison, avec  $t' \geq t$
- ▶ instruction instantanée si  $t' = t$
- ▶ instant suivant :  $t + 1$

## ▶ Signal

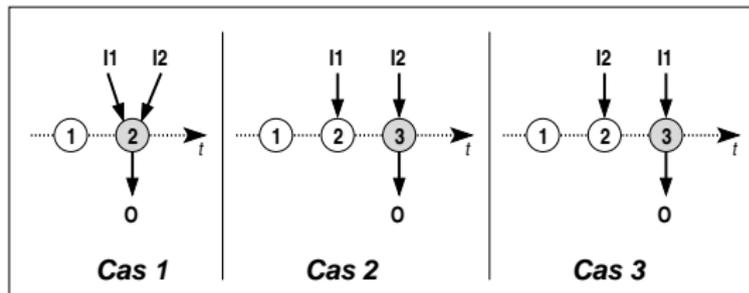
- ▶ en entrée et/ou en sortie
- ▶ caractérisé par son statut : présent ou absent
- ▶ pur ou valué (information typée)
- ▶ généré par l'environnement ou envoyé avec `emit`
- ▶ diffusé *instantanément* (*broadcast*)
- ▶ tick : top d'horloge

# Cycle de vie d'une instruction

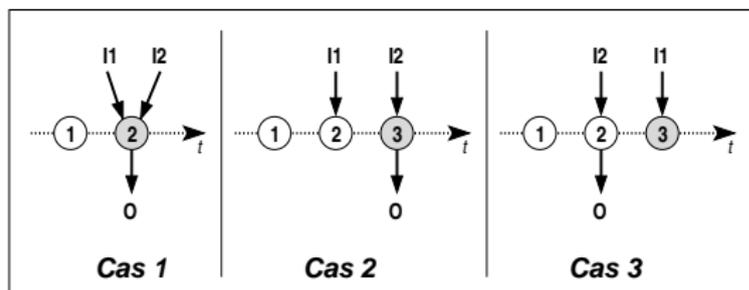
- ▶ Terminaison
  - ▶ spontanée, sauf pour la boucle infinie (`loop`)
  - ▶ avortement, si dans une construction `abort`
  - ▶ composition parallèle : quand les 2 composantes ont terminé
  
- ▶ Suspension = composition active sur plusieurs instants
  - ▶ `pause` : reprise à l'instant suivant
  - ▶ `await` : attente d'un signal

# Premiers exemples

```
[ await I1 || await I2 ] ; emit 0
```



```
await I1 || await I2 ; emit 0
```

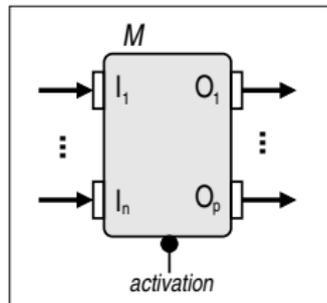


# Modules

- ▶ Programme Esterel = module
  - ▶ interface : signaux en entrée/sortie + déclarations d'objets externes
  - ▶ corps : composition d'instructions impératives et réactives
  - ▶ sous-modules

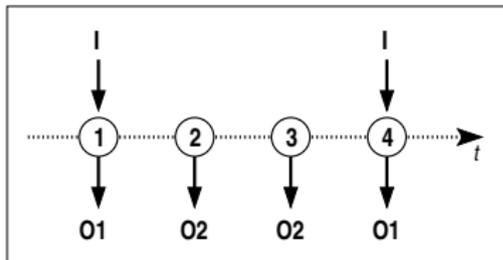
- ▶ Réaction d'un module

- ▶ réaction instantanée à l'activation
- ▶ émission de signaux et/ou modifications internes



## Exemple de module

```
module M :  
  % Interface  
  input I ;  
  output O1, O2 ;  
  
  % Corps  
  loop  
    present I then  
      emit O1  
    else  
      emit O2  
    end present ;  
    pause  
  end loop  
end module
```



**Attention à la boucle infinie !**

## Expression de signaux

▶ Instructions liées au statut : `present` et `await`

▶ Attente infinie : `halt`

▶ Attente de plusieurs occurrences

```
await 4 S
```

▶ Expressions booléennes

```
present [ I1 and I2 ] or [ not I3 ]  
  then emit 0  
end present
```

## Émission instantanée de signal

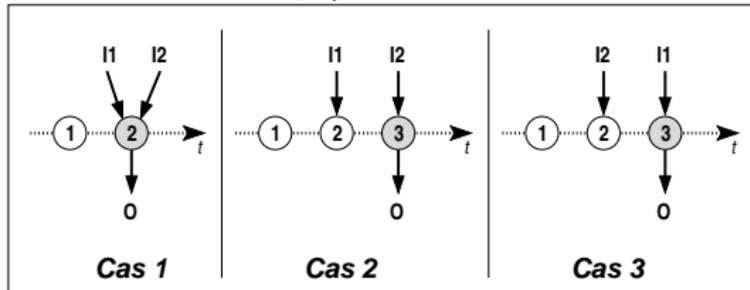
- ▶ Avantage : partage immédiat du statut et de la valeur du signal aux composantes parallèles et aux sous-modules
- ▶ Inconvénient : la présence d'un signal peut être testée dans l'instant même où il est émis ...

```
% 0 sera émis à chaque instant !  
loop  
  [ present S  
    then emit 0  
    end present  
  ||  
    emit S ] ; pause  
end loop
```

## Immédiat vs. différé

- ▶ `await` = attente différée à l'instant suivant

```
[ await I1 || await I2 ] ; emit 0
```



- ▶ Prise en compte immédiate d'un signal

```
await immediate I ;  
emit 0
```

```
if present I  
  then emit 0  
  else await I ; emit 0  
end present
```

# Manipulation de données

## ▶ Types

- ▶ prédéfinis : `integer`, `float`, `double`, `boolean`, `string`
- ▶ externes/utilisateurs

## ▶ Variable locale

- ▶ déclaration : `var V : integer in ... end var`
- ▶ lecture de la valeur : `V`
- ▶ affectation : `V := 1`
- ▶ déclaration + initialisation : `var V := 0 : integer in...`

## ▶ Attention aux compositions invalides !

```
% Doit être refusé par le compilateur  
X := X + 1 || X := 3
```

## Exemple d'utilisation de variables

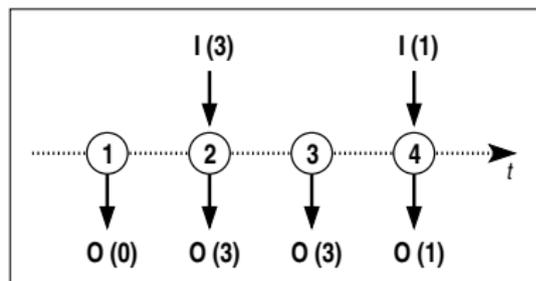
```
module M :  
  output Equal ;  
  constant C : integer ;  
  var X := 0 : integer, Y : integer in  
    Y := C ;  
    loop  
      if (X = Y)  
        then emit Equal ; X := 0 ; Y := C  
        else X := X + 1  
        end if ;  
      pause  
    end loop  
  end var  
end module
```

# Valeur d'un signal (1)

- ▶ Valeur indéterminée tant que le signal n'a pas été émis
- ▶ Opérations
  - ▶ déclaration : `input I : boolean ;`
  - ▶ déclaration + initialisation : `input I := true : boolean ;`
  - ▶ lecture de la dernière valeur : `?I`
  - ▶ émission évaluée : `emit 0 (13)`
- ▶ Attention à la diffusion instantanée !  
`emit S (?S + 1) % Impossible d'avoir S = S+1`

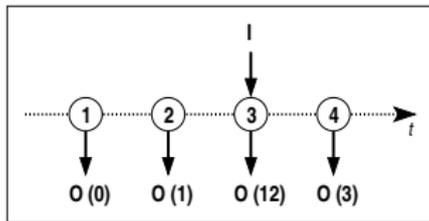
## Valeur d'un signal (2)

```
module Observe :  
  input I := 0 :  
    integer ;  
  output O :  
    integer ;  
  
  loop  
    emit O (?I) ;  
    pause  
  end loop  
end module
```



# Combinaison de valeurs d'un signal

Quand un signal valué peut être émis plusieurs fois au même instant



```
module M :
  input I ;
  output O :
    combine integer
    with + ;
  var X := 0 : integer
  in
    loop
      [ emit O (X) ; X := X + 1
        || present I then emit O (10) ] ;
      pause
    end loop
  end var
end module
```

# Capteurs

Signal dégénéré sans statut  $\Rightarrow$  variable externe en lecture seule

```
module Thermometre :  
  input Calculate ;  
  output Fahrenheit : float ;  
  sensor Celsius : float ;  
  
  function c2f (float) : float ;  
  
  loop  
    emit Fahrenheit (c2f (?Celsius)) ;  
  each Calculate  
end module
```

# Types, fonctions et procédures externes

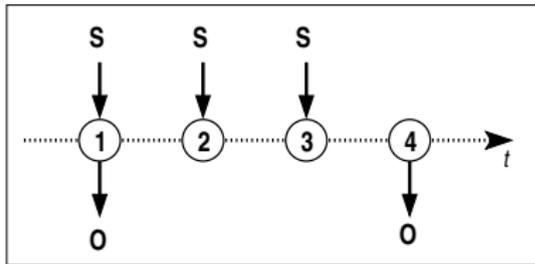
Les fonctions et les procédures s'exécutent en temps nul !

```
module M :
  type T ;
  procedure Increment (T) (int) ; % E/S et E
  function Init () : T ;
  function Test (T) : boolean ;
  var X := Init () : T
  in
    loop
      if (Test (X))
        then call Increment (X) (1)
      end if ;
      pause
    end loop
  end var
end module
```

# Préemption

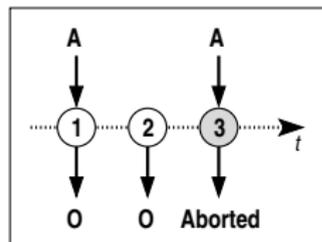
- ▶ Trois mécanismes pour interrompre une composition sur un signal :
  - ▶ suspension
  - ▶ avortement fort
  - ▶ avortement faible
- ▶ Suspension... puis reprise

```
suspend
loop
  emit 0 ;
  pause
end loop
when S
```

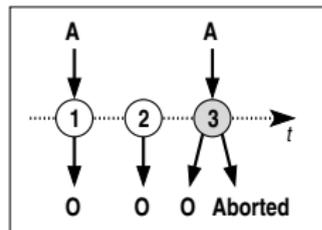


# Avortement fort ou faible ?

```
abort
  loop
    emit 0 ; pause
  end loop
when A ;
emit Aborted
```



```
weak abort
  loop
    emit 0 ; pause
  end loop
when A ;
emit Aborted
```



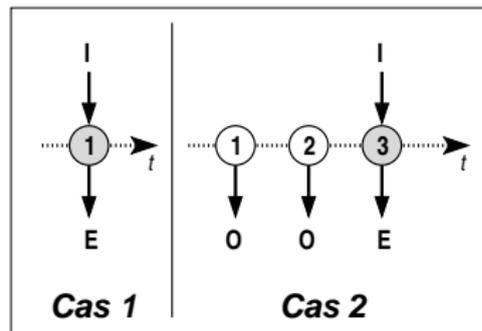
+ mot-clé `immediate`

+ composition à exécuter quand le signal est reçu (`do...end abort`)

# Trappes

S'échapper d'un traitement en précisant un motif

```
trap T in
  loop
    present I
      then exit T
    end present ;
    emit 0
  each tick
end trap ;
emit E
```



+ `handle...do` pour récupérer l'échappement

# Instructions dérivées (1)

## ▶ Boucles étendues

### ▶ `loop p each S`

si  $p$  termine, attendre  $S$  pour reprendre

si  $S$  reçu avant la terminaison de  $p$ , avorter  $p$  puis reprendre

```
loop          loop
  emit 0      abort
  each I      emit 0 ; halt
              when S
              end loop
```

### ▶ `every S do p`

idem, sauf attente d'une première occurrence de  $S$

## ▶ Répétition instantanée :

`repeat n times p end repeat`

## Instructions dérivées (2)

- ▶ Test multiple de signaux vs. attente hiérarchisée

```
present
  case Hello do
    emit 0 ("hello")
  case World do
    emit 0 ("world")
  else
    emit 0 ("snif!")
end present
```

```
await
  case I1 do
    emit 0 (1)
  % Si I1 absent...
  case I2 do
    emit 0 (10)
  % Pas de 'else' !
end await
```

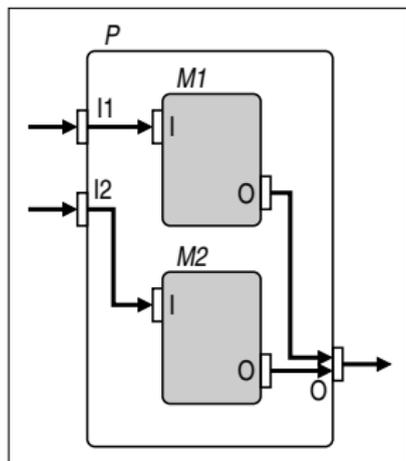
- ▶ Émission permanente d'un signal

```
sustain 0    <=>
```

```
loop
  emit 0
each tick
```

# Composition de modules (1)

Instanciation de sous-modules + substitution de signaux



```
module M :  
  input I ;  
  output O ;  
  % Corps...  
end module
```

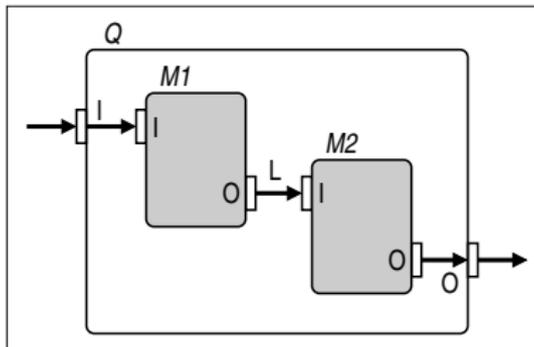
```
module P :  
  input I1, I2 ;  
  output O ;
```

```
    run M1/M [ signal I1/I ]  
  ||  
    run M2/M [ signal I2/I ]  
end module
```

## Composition de modules (2)

+ déclaration de signaux locaux

```
module M :  
  input I ;  
  output O ;  
  % Corps...  
end module  
  
module Q :  
  input I ;  
  output O ;  
  signal L in  
    run M1/M [ signal L/O ]  
  ||  
    run M2/M [ signal L/I ]  
  end signal  
end module
```



# Tâches asynchrones (1)

- ▶ Pour les traitements non-instantanés = transformationnels  
⇒ traitements externes
- ▶ Manipulation de tâche :
  - ▶ Déclaration d'une tâche : `task`
  - ▶ Déclaration d'un signal de retour : `return`
  - ▶ Lancement de l'exécution d'une tâche : `exec`
  - ▶ puis attente implicite de la fin d'une tâche
- ▶ Terminaison spontanée
  - ▶ mise à jour instantanée des paramètres
  - ▶ instruction `exec` terminée
- ▶ Test du signal de retour : fin spontanée ou avortement ?

## Tâches asynchrones (2)

```
module M :
  type Coords, Traj ;
  input Current : Coords ;
  output NewTrajectory : Traj ;
  return R ;
  task ComputeTrajectory (Traj) (Coords) ;
  var T : Traj in
    [ loop
      await Current ;
      exec ComputeTrajectory (T) (?Current)
        return R ;
      emit NewTrajectory (T)
    end loop ]
  || p % Corps à exécuter en parallèle
end var
end module
```

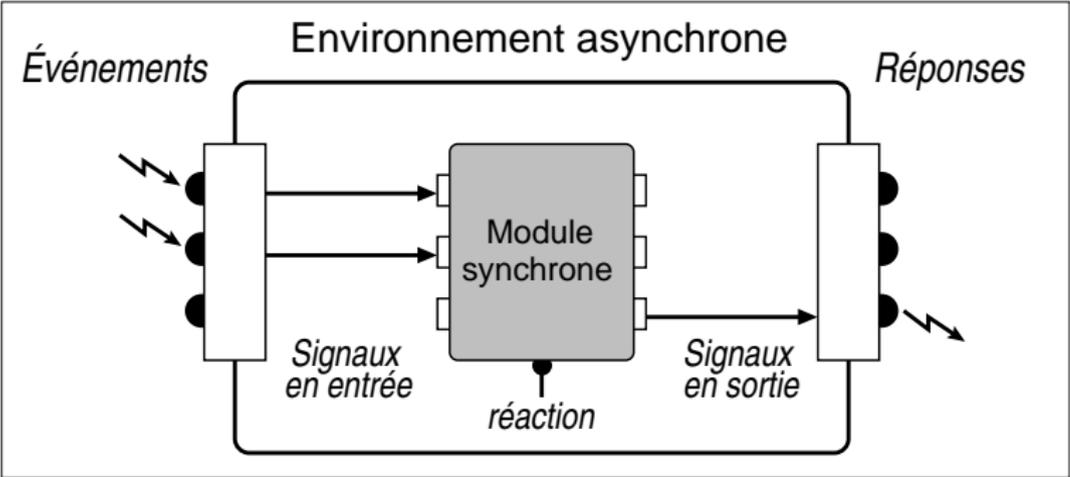
# Exécution d'un système décrit en Esterel

- ▶ Machine d'exécution
- ▶ Compilation d'un programme Esterel

# Machine d'exécution (1)

- ▶ Mécanismes d'interfaçage système/environnement
  - ▶ stockage des événements asynchrones
    - ⇒ traduction en signaux synchrones
  - ▶ activation du module
  - ▶ exécution des tâches asynchrones
- ▶ Contraintes :
  - ▶ exécution complète de chaque réaction
  - ▶ positionnement des signaux en entrée avant l'activation
- ▶ Stratégies d'activation
  - ▶ sur l'arrivée des signaux (événements sporadiques)
  - ▶ par échantillonnage périodique (interaction continue)

# Machine d'exécution (2)

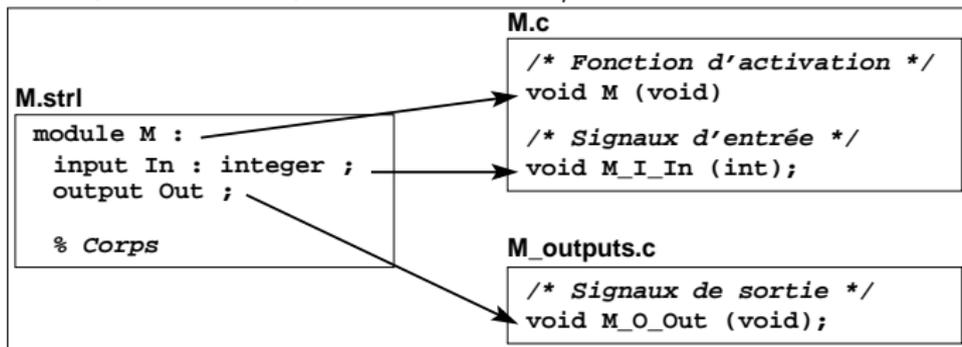


# Compilation d'un programme Esterel

- ▶ Vers la description :
  - ▶ d'un automate
  - ▶ d'un circuit booléen  $\Rightarrow$  circuit simulé ou réel
- ▶ Étapes de compilation en C :

```
str1 m.str1
gcc -c m.c m_main.c
gcc -o m.m.o m_main.o
```

- ▶ Exemple de correspondance Esterel/C

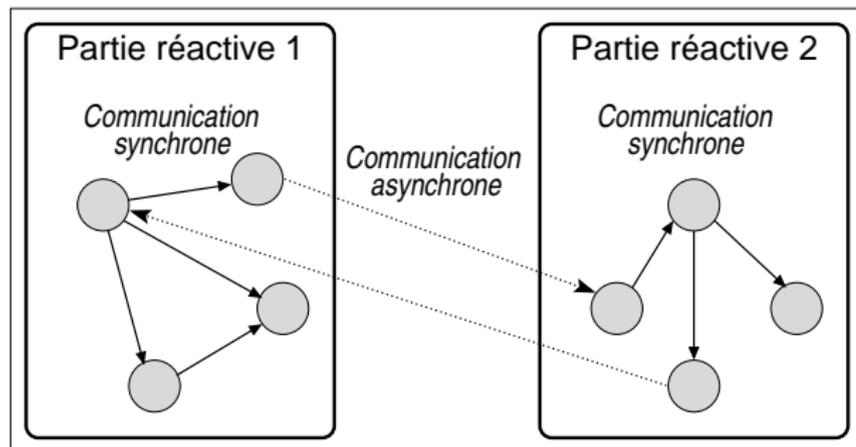


# Conclusion

- ▶ Modèle synchrone
  - ▶ facilite la description de systèmes réactifs
  - ▶ comportement déterministe garanti
  - ▶ vérification formelle
- ▶ Esterel
  - ▶ langage impératif synchrone
  - ▶ adapté à la description de systèmes temps-réel critiques
  - ▶ nombreux outils de développement disponibles
  - ▶ nécessite une machine d'exécution respectant les hypothèses du modèle synchrone
  - ▶ inconvénient : architecture statique du système décrit

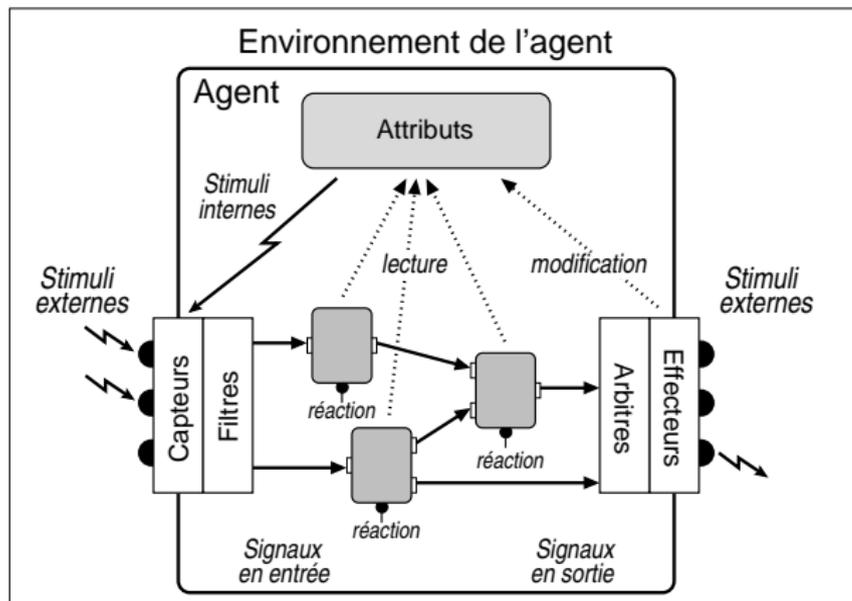
## Autres applications du modèle synchrone (1)

- ▶ F. Boussinot
  - ▶ simplification : réaction à l'absence d'un signal reportée au début de l'instant suivant
  - ▶ Reactive-C, FairThreads, réseaux de processus réactifs, object réactifs, ...
- ▶ F. Boulanger : objets synchrones



## Autres applications du modèle synchrone (2)

- Description d'agents virtuels :  
modèle InViWo et langage Marvin



## Références

*The Foundations of Esterel*, G. Berry, 1998.

*The Esterel v5 Language Primer*, G. Berry, 1999.

*On the development of reactive systems*, D. Harel et A. Pnueli, 1985.

[www.esterel-technologies.com/](http://www.esterel-technologies.com/)

[www-sop.inria.fr/meije/meije-fra.html](http://www-sop.inria.fr/meije/meije-fra.html)

[www-sop.inria.fr/mimosa/rp/](http://www-sop.inria.fr/mimosa/rp/)

## Exemple : le réveil (1)

```
module reveil_matin :  
  
    input Minute ;  
    input AlarmAt : integer ; % En minutes  
    input CancelAlarm ;  
  
    output WakeUp ;  
    output Time : integer ; % En minutes
```

## Exemple : le réveil (2)

```
% Écoulement des minutes
var elapsed : integer in
  elapsed := 0 ;
  every Minute do
    elapsed := elapsed + 1 ;
    emit Time (elapsed) ;
  end every
end var
```

||

## Exemple : le réveil (3)

```
% Gestion de l'alarme
every AlarmAt do
  abort
  await ?AlarmAt Minute ;
  emit WakeUp ;
  when CancelAlarm ;
end every
end module
```

## Exemple : le téléphone (1)

```
module telephone :  
  input Seconde ;  
  input Decrocher ;  
  input Saisie_numero ;  
  input Appel ;  
  input Raccrocher ;  
  
  output Temps_communication : integer ;  
  output Sonnerie ;  
  output Echec_appel ;
```

## Exemple : le téléphone (2)

```
% Appel sortant
loop
  var echec : boolean in
    await Decrocher ;
    echec := false ;
  abort
    await 10 Seconde ;
    emit Echec_appel ;
    echec := true ;
  when Saisie_numero ;
```

## Exemple : le téléphone (3)

```
if not echec then
  abort
  var total := 0 : integer in
    every Seconde do
      total := total + 1 ;
      emit Temps_communication (total)
    end every
  end var
  when Raccrocher ;
  else await Raccrocher ;
end if
end var
end loop
```

## Exemple : le téléphone (4)

```
|| % Appel entrant
loop
  var echec : boolean in
    await Appel ;
    echec := false ;
    abort
      abort
        every Seconde do
          emit Sonnerie ;
        end every;
      when 20 Seconde ;
        emit Echec_appel ;
        echec := true ;
    when Decrocher ;
```

## Exemple : le téléphone (5)

```
if not echec then
  abort
  var total := 0 : integer in
    every Seconde do
      total := total + 1 ;
      emit Temps_communication (total) ;
    end every
  end var
  when Raccrocher ;
end if
end var
end loop
end module
```