

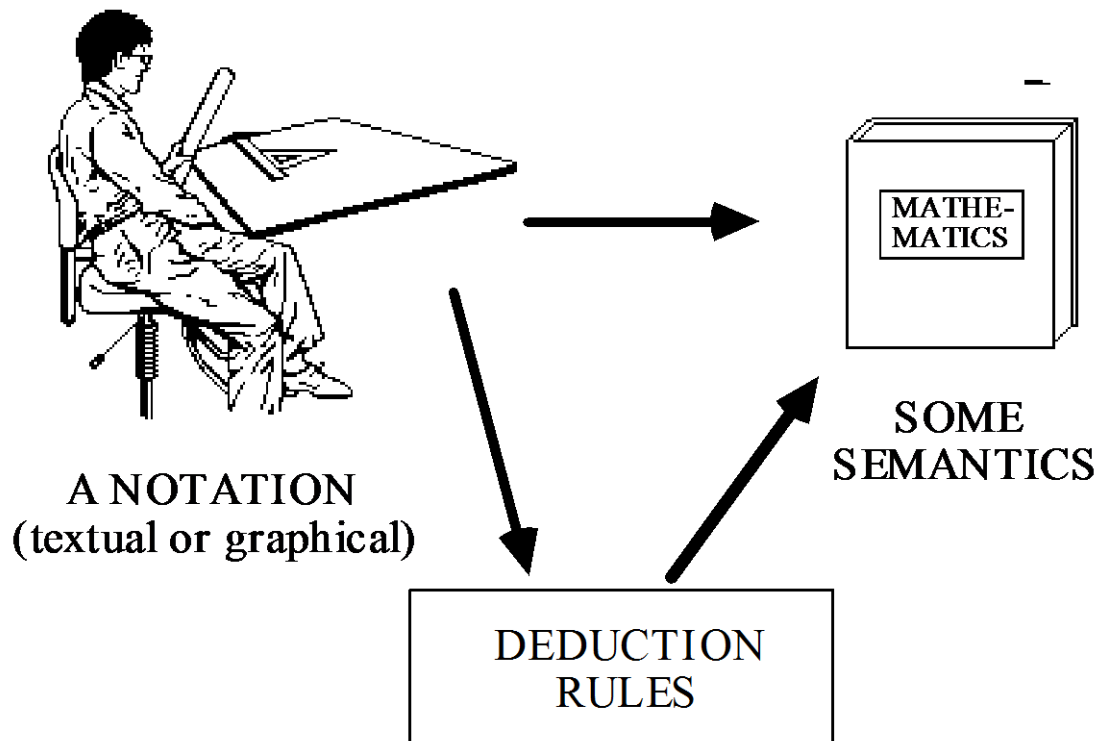
# LES LANGAGES DE SPECIFICATION – GENERALITES SUR LE LANGAGE LOTOS

## 2.1 - Langages de spécification formelle

Une méthode formelle de développement de logiciel consiste en :

- une *notation* pour la spécification et le développement de logiciel ;
- avec une *signification mathématique* ;
- un *système formel* de raisonnement qui permet de faire des preuves ou du calcul symbolique ;
  - une *relation de raffinement (ou d'implémentation)* entre spécifications ;

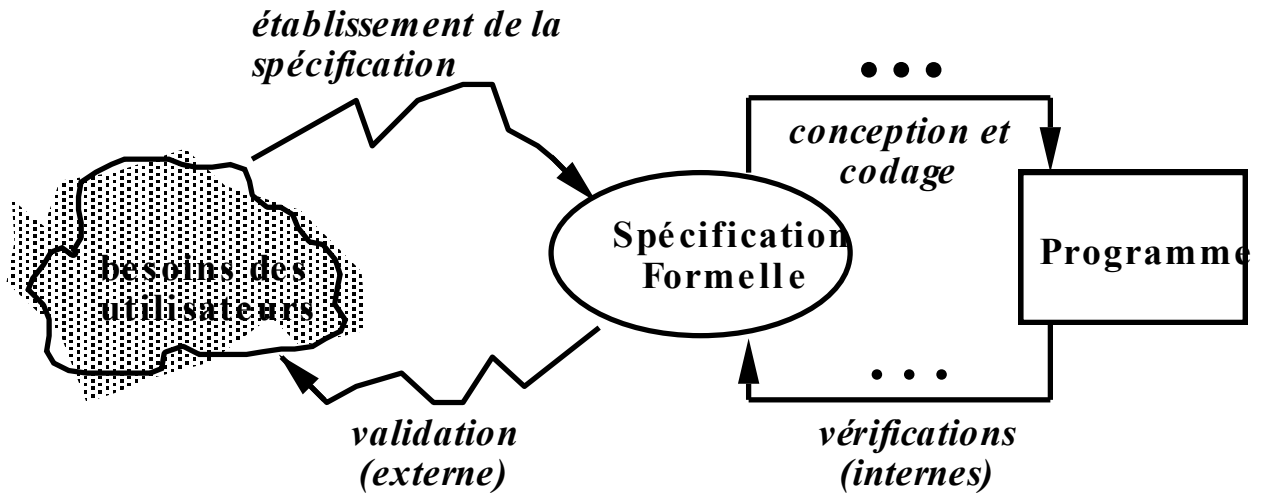
- et une *relation de satisfaction* entre spécification détaillée et programme.



### exemple de LOTOS :

- Sémantique : automates (finis ou non)
- Dédution : raisonnement sur le graphe de l'automate ou l'arbre des comportements

# UN MODELE DE DEVELOPPEMENT BASE SUR UNE METHODE FORMELLE



*Conception/Raffinements/Implémentations*

$SP_0 \rightarrow SP_1 \rightarrow \dots \rightarrow SP_n$

*Codage/Programmation*

$SP_n \rightarrow \text{Prog}$

exemple de LOTOS : génération de code C

*Vérifications*

$SP_0 \leftarrow SP_1 \leftarrow \dots \leftarrow SP_n \leftarrow \text{Prog}$

Vérification "bi-langage" :  $SP_n \leftarrow \text{Prog}$

## VALIDATION ET SPECIFICATIONS FORMELLES

- *La spécification est-elle “adéquate”?*  
pas de référence absolue de correction, mais :
  - vérification de la consistance interne
  - preuve ou réfutation de “challenges”
  - animation et test de la spécification

## VERIFICATIONS ET SPECIFICATIONS FORMELLES

- *Chaque étape préserve-t-elle les propriétés requises par la précédente?*  
référence de correction : la spécification de départ
  - preuve que  $SP_i$  est un raffinement (implémentation) correct de  $SP_{i-1}$
  - preuve du programme par rapport à sa spécification détaillée
  - test du système par rapport aux spécifications

## 2.2 - Introduction à LOTOS : Lotos de Base et Lotos Complet

### LOTOS

*“Language Of Temporal Ordering Specification”*, ISO, 1988

Question subsidiaire : *qu'est-ce que l'ISO?*

LOTOS est un standard utilisé pour spécifier les protocoles de télécommunication

Utilisable pour les systèmes distribués en général

Permet de décrire “des ensembles de processus qui interagissent et échangent des données entre eux et avec leur environnement”

<http://www.cs.stir.ac.uk/~kjt/research/well>

<http://www.inrialpes.fr/vasy> (CADP)

<http://www-run.montefiore.ulg.ac.be/Research>  
(EUCALYPTUS)

<http://www.csi.uottawa.ca/~lotos> (ELUDO)

<http://www.tios.cs.utwente.nl/lotos> (SMILE)

## PREMIER EXEMPLE (ASSEZ NUL) EN LOTOS COMPLET

---

```

specification Exemple : noexit
  library NaturalNumber endlib
  behavior
    P1[port] | [port] P2[port]
  where
    process P1 [port] : noexit :=
      port ! 0 ; P2[port]      (* P1 envoie 0 puis se
comporte                       comme P2 *)
    endproc (* P1 *)
    process P2 [port] : noexit :=
      port ? x : Nat ; port ! succ(x) ; P2[port] (* P2 reçoit x,
envoie x+1, et recommence ... *)
    endproc (* P2 *)
  endspec (* Exemple *)

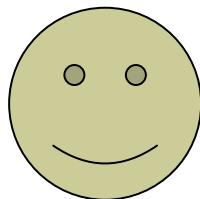
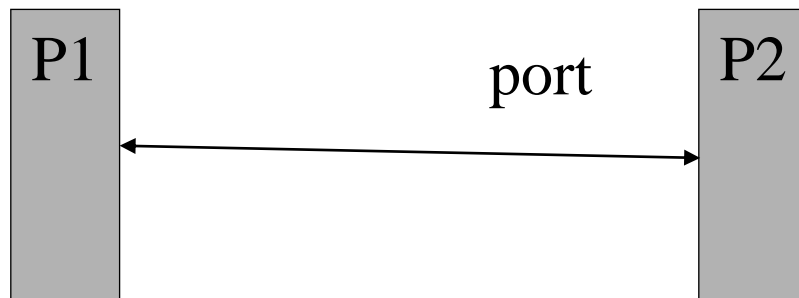
```

---

- Il y a des types de données : NaturalNumber
- Il y a des ports de communication : port
  - Il y a des processus : P1, P2
- Il y a des actions : port ! 0, port ? x : Nat , port ! succ(x)

*Que fait le système spécifié?*

# ARCHITECTURE DU SYSTEME SPECIFIE



Il ne présente aucun intérêt...

## Une variante non déterministe

---

```

process P2 [port] : noexit :=
  port ? x : Nat ;
  ( [ x le 1000 = true ] ->(port ! succ(x) ; P2[port] )
  [ ]
  ( [ x ge 1 = true ] ->(port ! pred(x) ; P2[port] )
endproc (* P2 *)

```

---

# LOTOS DE BASE

## (Basic Lotos)

Version simplifiée

*Pas de types de données*

Alphabet fini d'actions observables :

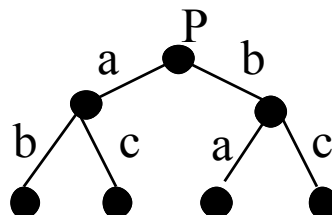
*action observable*  $\Leftrightarrow$  *port*

Les processus se synchronisent, mais ne communiquent pas

Intérêt : sémantique simple, par des automates finis et des arbres réguliers

```

process P[a, b, c] :=
  (a ; ( b; stop [ ] c; stop )
  [ ]
  (b ; (a ; stop [ ] c ; stop))
endproc
  
```



Problème : pouvoir d'expression trop limité

## LOTOS COMPLET

On peut définir des types de données :  
*par exemple, Message, File de Messages, ...*

Les ports servent à communiquer des valeurs :  
**in ? m : Message,**  
**out ! first (buffer)**

On a des “gardes” i. e. des choix  
conditionnels :

**([isEmpty(buffer) = false] → out ! first (buffer); ...)**  
**[**  
**(in ? m : Message; ...)**

Sémantique plus complexe : automates et  
arbres avec arités infinies (nb infini de  
branches à un noeud)

Les processus peuvent être paramétrés et donner des résultats :  $\pm$  coopération par variables partagées

## 2.3 - LES TYPES DE DONNEES EN LOTOS :

Spécification Algébrique des Types de Données

Description des propriétés requises, indépendante de la future implémentation

Signature + Axiomes
---------------------

**type Boolean is**

**sort Bool**

**opns**

true, false :  $\rightarrow$  Bool

not : Bool  $\rightarrow$  Bool

**eqns**

**ofsort Bool**

not(true) = false

not(false) = true

**endtype**

---

## FORME GENERALE DE LA DEFINITION D'UN TYPE

---

<b>type</b> T is T1, ..., Tn	<i>(* types utilisés *)</i>
<b>sorts</b> ...	<i>(* sortes définies*)</i>
<b>opns</b>	<i>(* noms des opérations et</i>
...	<i>leurs profils *)</i>
<b>eqns</b>	
<b>forall</b> ...	<i>(* déclarations des</i>
	<i>variables *)</i>
<b>ofsort</b> ...	
...	<i>(* équations d'une même</i>
...	<i>sorte *)</i>
<b>ofsort</b> ...	
...	<i>(* etc *)</i>
<b>endtype</b>	

---

NB : importante bibliothèque de types  
prédéfinis

### Les axiomes de Lotos :équations conditionnelles

*equation = [premisses “=>” ] simple-equation*

*premisses = premiss {“,” premiss}*

*premiss = simple-equation | boolean-expression*

*simple-equation = value-expression “=” value-expression*

## UN EXEMPLE PLUS SOPHISTIQUE

---

```
type FileNat is NaturalNumber
sorts File
opns
  vide :      → File
  ajouter : File, Nat → File
  retirer :   File → File
  premier :  File → Nat
eqns
  forall x, y : Nat, z : File
  ofsort Nat
    premier (ajouter(vide, x)) = x
    premier (ajouter(ajouter(z, x), y) =
              premier(ajouter(z, x))
    (* quid de premier(vide)? *)
  ofsort File
    retirer (ajouter(vide, x)) = vide
    retirer (ajouter(ajouter(z, x), y) =
              ajouter(retirer(ajouter(z, x)), y)
endtype (* FileNat *)
```

---

## Exemple d'équations conditionnelles

**forall**  $x, y : \text{Nat}, z : \text{File}$

**ofsort**  $\text{Nat}$

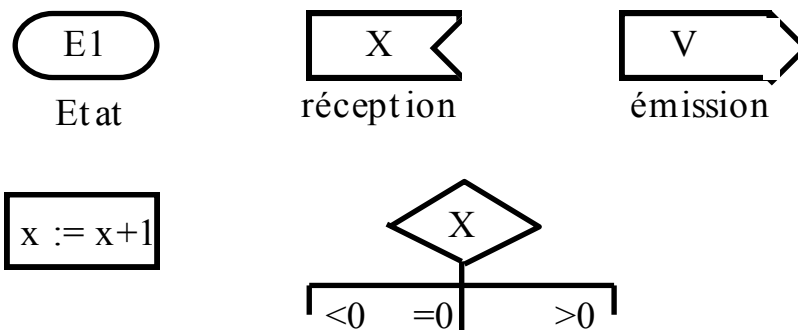
$\text{est-vide}(z) \Rightarrow \text{premier}(\text{ajouter}(z, x)) = x$  ;

$\text{est-vide}(z) = \text{false} \Rightarrow \text{premier}(\text{ajouter}(z, x)) = \text{premier}(z)$  ;

## CONCURRENT DE LOTOS : SDL (ou LDS en Français)

“Specification and Description Language”  
ISO, 92

Notation graphique :



Proche d'un langage de programmation : on peut, au choix, spécifier des types ou utiliser ceux du langage d'implémentation

Communication asynchrone (buffers implicites)

=> sémantique plus compliquée,  
vérifications difficiles

# LOTOS DE BASE

## 1 Processus et ports

- un système est spécifié par :
  - des *définitions de processus*
  - et une *expression de comportements* qui utilise ces processus
- un processus peut exécuter
  - des *actions internes*
  - des interactions avec d'autres processus via des *ports* (“gates”)
- un processus est caractérisé par
  - son *nom*
  - les *nom de ses ports*
  - la *spécification de ses comportements*

**exemple :**  $P[a, b, c] := a; b; c; \text{stop}$

- un processus est générique/ses ports :

$P[d, d, a]$  a pour comportement  $d; d; a; \text{stop}$

## 2 Un exemple incomplet

---

```

process Max3[in1, in2, in3, out] :=
  hide mid in
    (Max2[in1, in2, mid]
      | [mid] |
      Max2[mid, in3, out])
  where
    process Max2[a, b, c] := ...
    ...
end proc (*Max3*)

```

---

Constructions utilisées :

- synchronisation sur mid : ...|[mid]|...
- masquage de mid qui est local à Max3 :  
**hide** mid in ...

Un peu de syntaxe :

---

```

process Max3[in1, in2, in3, out] := ... end proc (*Max3*)

```

---

est une définition de processus.

---

```

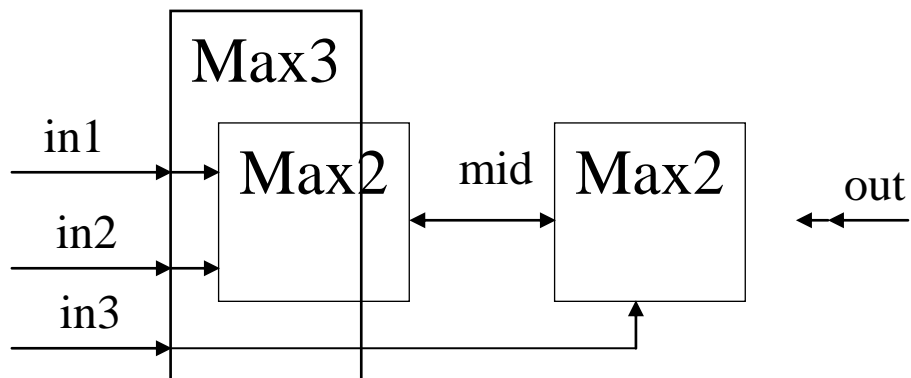
hide mid in
  (Max2[in1, in2, mid]
    | [mid] |
    Max2[mid, in3, out])
where process Max2[a, b, c] := ...
  ...

```

---

est une expression de comportements

## Architecture du Système



## Le même plus complet

---

```
process Max3[in1, in2, in3, out] :=  
  hide mid in  
    (Max2[in1, in2, mid]  
      | [mid] |  
      Max2[mid, in3, out])  
  where  
    process Max2[a, b, c] :=  
      a; b; c; stop  
      [  
      b; a; c; stop  
    ]  
    end proc (*Max2*)  
end proc (*Max3*)
```

---

Constructions utilisées :

- choix : [ ]
- généricité/ports de Max2

### 3 - Comment spécifier des comportements?

#### *La syntaxe en 1 page*

- B1, B2, ... expressions de comportements
- g1, g2, ... noms de ports
- P, Q, ... noms de processus
  
- stop *arrêt définitif*
- i *action interne*
- g; B i; B *préfixe* : une action précède des comportements
- B1 [] B2 *choix // pbs ...*
- B1 ||| B2 *parallélisme sans synchro*
- B1 || B2 *parallélisme avec synchro totale*
- B1 | [g1, ..., gn] | B2 *parallélisme avec synchro partielle*
- **hide** g1, ..., gn **in** B *masquage*
- P[g1, ..., gn] *instantiation (on est dans la portée de la définition de P)*
- exit *terminaison avec succès*
- B1 >> B2 *composition séquentielle*
- B1 [> B2 *interruption de B1 par B2*

## Les pièges connus

stop et exit  
; et >>

## La sémantique

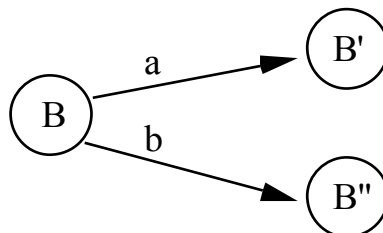
*Une expression de comportements correspond  
à un automate*

### notation

$B \xrightarrow{a} B'$

“On peut passer de l'état B à l'état B' par  
l'action a”

### exemple



correspond à l'expression  $B = a; B' [ ] b; B''$

***ATTENTION*** : *B* dénote à la fois une expression de comportements et son état initial

### Sémantique de “stop”

“stop”

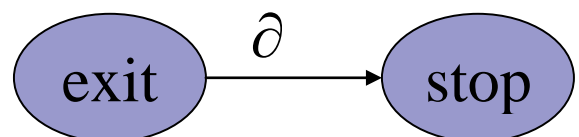


*On reste définitivement bloqué dans cet état*

Quoiqu’il arrive dans l’environnement, le système est arrêté

### Sémantique de “exit”

“exit - $\partial$ -> stop”

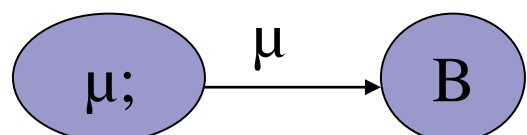


On peut exécuter l’action  $\partial$

On peut se synchroniser avec un autre processus

### Sémantique de $\mu; B$

“ $\mu; B$  - $\mu$ -> B”



( $\mu$  est soit un nom de port, soit  $i$ )

**exercice** : automates associés à :

$a$ ; **stop**       $i$  ; **exit**

**Sémantique du choix**

$B1 [ ] B2 \text{ -?-> ?}$

**Règles :**

$B1 \text{ -}\mu\text{-> } B'1 \Rightarrow B1 [ ] B2 \text{ -}\mu\text{-> } B'1$
$B2 \text{ -}\mu\text{-> } B'2 \Rightarrow B1 [ ] B2 \text{ -}\mu\text{-> } B'2$

- on fait l'union des transitions possibles en  $B1$  et  $B2$
- une fois le choix fait, *on oublie complètement l'autre choix*

**exemple** :  $a; b; c; \text{stop} [ ] b; a; c; \text{stop}$

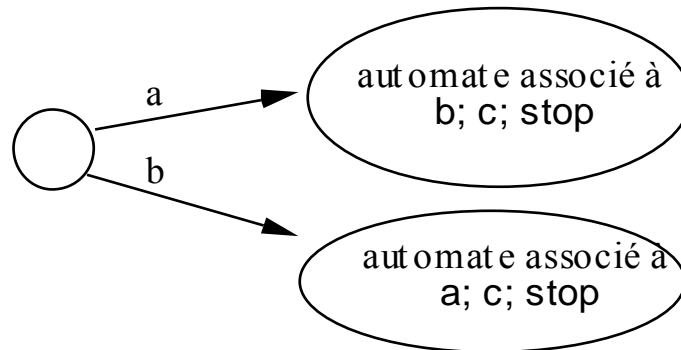
comme on a  $a; b; c; \text{stop} \text{ -}a\text{-> } b; c; \text{stop}$

on a  $a; b; c; \text{stop} [ ] b; a; c; \text{stop} \text{ -}a\text{-> } b; c; \text{stop}$

On a aussi

$a; b; c; \text{stop} [ ] b; a; c; \text{stop} \text{ -}b\text{-> } a; c; \text{stop}$

Pourquoi ?



**une erreur à ne pas faire :**

~~(a ; b [ ] b ; a) ; c ; stop ☹~~

C'est TRÈS faux :

- *syntactiquement*, devant “;” il faut une action et ici on a une expression de comportements

Par contre, on peut écrire :

c; (a;b;stop [ ] b;a;stop)

qui ne donne pas le même automate que

c;a;b;stop [ ] c;b;a;stop

!!! *on y reviendra*

## Sémantique du masquage

**hide g in B**

On remplace “g” par “i” dans l’automate associé à B

## Sémantique de l’instantiation

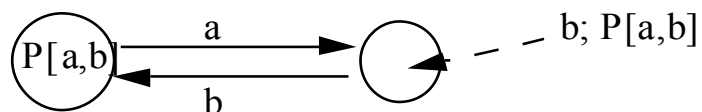
$P[g_1, \dots, g_n]$

P est défini par  $P[x_1, \dots, x_n] := B \Rightarrow$   
on remplace les  $x_i$  par  $g_i$  dans l’automate associé à B

**Récurtivité  $\Rightarrow$  boucle dans l’automate**

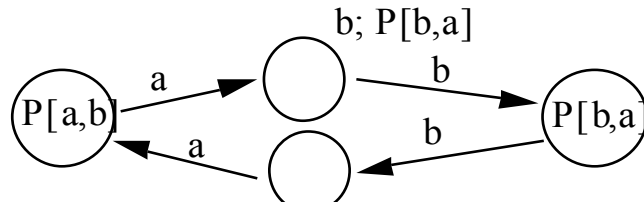
1- cas direct

$P[a,b] := a;b; P[a,b]$



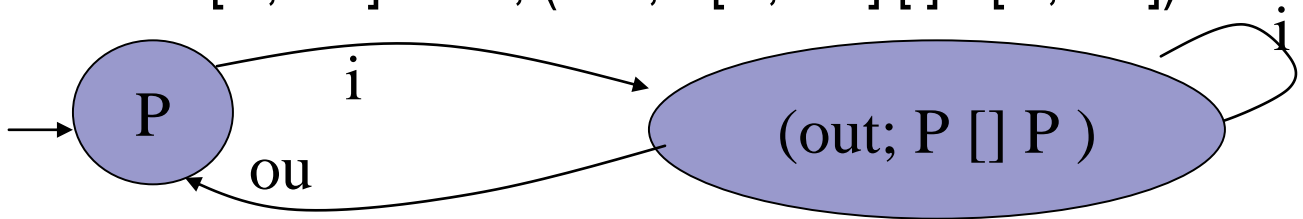
2-cas moins direct

$P[a,b] := a;b; P[b,a]$



3- cas encore moins direct

$P[\text{in}, \text{out}] := \text{in} ; (\text{out} ; P[\text{in}, \text{out}] [] P[\text{in}, \text{out}])$



# SEMANTIQUE DU PARALLELISME

## 1- sans synchronisation

$$B1 \parallel B2$$

### Règles :

( $\mu$  est un port ou  $i$ )

$$B1 \xrightarrow{-\mu} B'1 \Rightarrow B1 \parallel B2 \xrightarrow{-\mu} B'1 \parallel B2$$

$$B2 \xrightarrow{-\mu} B'2 \Rightarrow B1 \parallel B2 \xrightarrow{-\mu} B1 \parallel B'2$$

$$B1 \xrightarrow{-\partial} B'1 \text{ et } B2 \xrightarrow{-\partial} B'2 \Rightarrow$$

$$B1 \parallel B2 \xrightarrow{-\partial} B'1 \parallel B'2$$

(on entrelace les actions externes ou internes de B1 et B2 ; on se synchronise sur la terminaison avec succès)

### exemple

---

```

process duplex_buffer1 [ina, inb, outa, outb] :=
  simplex_buffer[ina, outa] |||
  simplex_buffer[inb, outb]
where
  process simplex_buffer[in, out] := in; out; exit
end proc
end proc

```

---

## 2- avec synchronisation totale

$B1 \parallel B2$

### Règles

( $\beta$  est un port ou  $\partial$ )

$B1 \text{ -}\beta\text{-> } B'1 \text{ et } B2 \text{ -}\beta\text{-> } B'2 \Rightarrow$

$B1 \parallel B2 \text{ -}\beta\text{-> } B'1 \parallel B'2$

$B1 \text{ -}i\text{-> } B'1 \Rightarrow B1 \parallel B2 \text{ -}i\text{-> } B'1 \parallel B2$

$B2 \text{ -}i\text{-> } B'2 \Rightarrow B1 \parallel B2 \text{ -}i\text{-> } B1 \parallel B'2$

(on se synchronise sur tout sauf  $i$ )

NB : on va bloquer souvent!

### exemples

$a; B \parallel b; B'$  bloque toujours.

$a; B \parallel (a; B' [ ] b; B'')$  exécute toujours “a” suivie de  $B \parallel B'$ , ne bloque jamais,  $B''$  n'est jamais exécutée.

$a; B \parallel (a; B' [ ] i; a; B'')$  exécute toujours “a” suivie soit de  $B \parallel B'$ , soit de  $B \parallel B''$ .

$a; B \parallel (i;a;B' \parallel a;B'')$  aussi,  $a; B \parallel (i;a;B' \parallel i;a;B'')$   
aussi, MAIS

$a; B \parallel (a;B' \parallel i;b;B'')$  peut bloquer,  
 $a; B \parallel (i;a;B' \parallel i;b;B'')$  aussi.

### 3 - avec synchronisation partielle

$B1 \parallel [g1, \dots, gn] \parallel B2$

#### Règles

$$(\beta \in \{g1, \dots, gn\} \cup \partial)$$

$$B1 \xrightarrow{\beta} B'1 \text{ et } B2 \xrightarrow{\beta} B'2 \Rightarrow$$

$$B1 \parallel [g1, \dots, gn] \parallel B2 \xrightarrow{\beta} B'1 \parallel [g1, \dots, gn] \parallel B'2$$

$$(\mu \notin \{g1, \dots, gn\} \cup \partial, \text{ ou } \mu=i)$$

$$B1 \xrightarrow{\mu} B'1 \Rightarrow B1 \parallel [g1, \dots, gn] \parallel B2 \xrightarrow{\mu} B'1 \parallel [g1, \dots, gn] \parallel B2$$

$$B2 \xrightarrow{\mu} B'2 \Rightarrow B1 \parallel [g1, \dots, gn] \parallel B2 \xrightarrow{\mu} B1 \parallel [g1, \dots, gn] \parallel B'2$$

(on se synchronise toujours sur  $\partial$ ,  
on ne se synchronise jamais sur  $i$ )

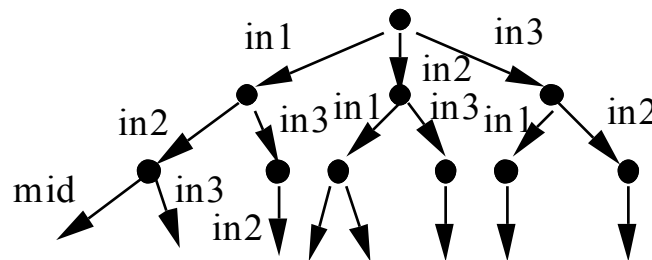
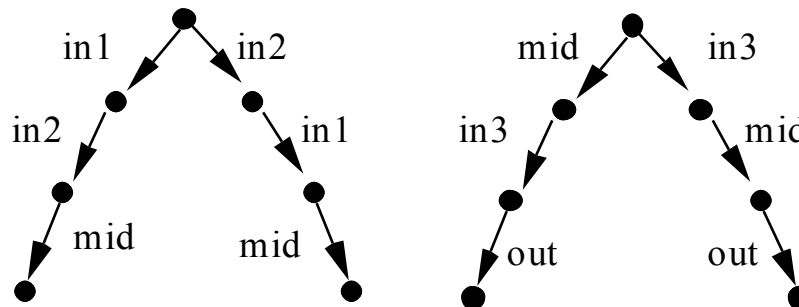
## exemple

Max2[in1, in2, mid] | [mid]|Max2[mid, in3, out]  
avec

Max2[a, b, c] := a; b; c; stop [ ] b; a; c; stop

Max2[in1, in2, mid] | [mid]|Max2[mid, in3, out]  
avec

Max2[a, b, c] := a; b; c; stop [ ] b; a; c; stop



## Parallélisme n-aire

Les trois constructions se généralisent pour  $n$   
*expressions de comportements:*

$$B1 \parallel \dots \parallel Bn$$
$$B1 \parallel \dots \parallel Bn$$
$$B1 \parallel [g1, \dots, gm] \dots \parallel [g1, \dots, gm] \parallel Bn$$

Les  $n$  processus doivent pouvoir exécuter la  
même action

# Sémantique de la composition séquentielle de COMPORTEMENTS

$B1 \gg B2$

## Règles

( $\mu$  est un port ou  $i$ )

$B1 \text{ -}\mu\text{-} \rightarrow B'1 \Rightarrow B1 \gg B2 \text{ -}\mu\text{-} \rightarrow B'1 \gg B2$

$B1 \text{ -}\partial\text{-} \rightarrow B'1 \Rightarrow B1 \gg B2 \text{ -}i\text{-} \rightarrow B2$

(on exécute  $B1$ ; *quand  $B1$  termine avec succès*, on exécute  $B2$ ):

mise en séquence des automates ou  $\partial$  indique où attacher l'état initial du 2ème)

## exemple

---

```

process duplex_buffer [ina, inb, outa, outb] :=
  duplex_buffer1 [ina, inb, outa, outb] >>
    duplex_buffer [ina, inb, outa, outb]
where
process duplex_buffer1 [ina, inb, outa, outb] :=
  simplex_buffer[ina, outa] ||| simplex_buffer[inb, outb]
where
process simplex_buffer[in, out] := in; out; exit
end proc
end proc
end proc

```

---

## autres exemples

(a;b;exit [ ] a;c;stop) >> P[...]

P est parfois exécuté (après a, b, i)

(a;b;exit ||| a;c;stop) >> P[...]

P n'est jamais exécuté. Pourquoi?

## Différences entre ; et >>

### 1 - syntaxe

<action> ; <expression de comportement>

<expression de comportement> >>

<expression de comportement>

### 2 - sémantique

- g;B ou i;B : B toujours exécutée après g ou i
- exit ; B : B jamais exécutée
- g >> B, ou i >> B : B jamais exécutée!
- exit >>B : B toujours exécutée!

## Sémantique de l'interruption

$B1 [ > B2$

### Règles

( $\mu$  est un port ou  $i$ )

$B1 -\mu-> B'1 \Rightarrow B1 [ > B2 -\mu-> B'1 [ > B2$

$B1 -\partial-> B'1 \Rightarrow B1 [ > B2 -\partial-> B'1$

$B2 -\mu-> B'2 \Rightarrow B1 [ > B2 -\mu-> B'2$

(chaque état de  $B1$ , est un état initial de  $B2$ )

### exemple

---

Activity[a, b, c] [ > Disrupt [d, r]

**where**

**process** Activity [a, b, c] := a; b; c; Activity [a, b, c]

**end proc**

**process** Disrupt [d, r] := d; r; **stop**

**end proc**

---

