Algèbres de processus (AdP)

AdP : formalisme théorique sur lequel s'appuient divers outils de spécification et de vérification.

Exemples d'AdP pour les systèmes asynchrones :

- **CCS** (*Calculus of Communicating Systems*) [Milner-89]
- **CSP** (*Communicating Sequential Processes*) [Hoare-85]
- **ACP** (*Algebra of Communicating Processes*) [Bergstra-Klop-84]

Langage normalisé : **LOTOS** [ISO-88]

Langage plus moderne : LOTOS NT (abrév. LNT)

Formalismes textuels (non graphiques)

- A la différence des AC et des RdP, les AdP ont une syntaxe textuelle (≈ programmes)

  $\Rightarrow$ même confrontation qu'entre programmes texte et organigrammes dans le domaine séquentiel
- Avantages :
  - plus de puissance d'expression (structuration)
  - passage à l'échelle (spécifications de grande taille)
- Inconvénients :
  - syntaxe moins intuitive

Approches pour spécifier un automate

- dessin (AC, RdP, langage SDL)
- énumération des états et des transitions (Estelle):

  state BUSY, IDLE, …

  trans from BUSY **to** IDLE

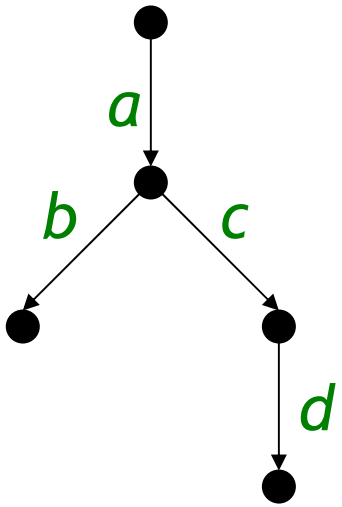
  x := x + 1

  Mécanismes similaires aux « goto »

  ⇒ manque de structuration, « spaghetti »
- utilisation des expressions régulières

Expressions régulières

- Théorie des langages : équivalence entre expressions régulières (ER) et automates finis
- Les AdP utilisent une variante des ER pour définir des automates
- Certaines équivalences de la théorie des langages ne sont plus valides : $a.(b + c) \neq a.b + a.c$



En CCS : *a*.(*b*.**nil** + *c*.*d*.**nil**)

En LNT :

a; **select** *b* [] *c*; *d* **end select**; **stop**

Expressions régulières (suite)

- Les AdP utilisent trois opérateurs similaires aux ER :
  - *choix* (‘+’ en CCS, ‘[]’ en CSP, ‘**select**’ en LNT), pour modéliser les branchements
  - *séquence* (‘ . ’ en CCS, ‘ ; ’ en CSP et LNT), pour modéliser les chaînes d’actions
  - *point fixe* (appel récursif de processus en CCS, idem et ‘**loop**’ en LNT), pour modéliser les circuits
- Avantage des ER : programmation structurée

Opérateurs de mise en parallèle

Les AdP possèdent un (ou plusieurs) opérateurs de parallélisme que l'on peut combiner librement avec les autres opérateurs (choix, séquence, etc.)

- En CCS : ' | '
- En CSP : ' || '
- En LNT : ' **par** G_1, …, G_n ' ($\approx \otimes_{\{G1, \ldots, Gn\}}$ des AC)

En AdP, on peut écrire (à peu près) :

a . (b | c) . d

Construction des comportements

- Les comportements complexes peuvent être décrits comme des expressions algébriques construites avec les opérateurs des AdP :

  $(a \mid b) \,.\, (c + d \,.\, (e \mid f))$

- L'idée des AdP : jeu de « lego » ; un petit nombre d'opérateurs primitifs, qui expriment chacun un concept et qu'on peut composer arbitrairement (orthogonalité)

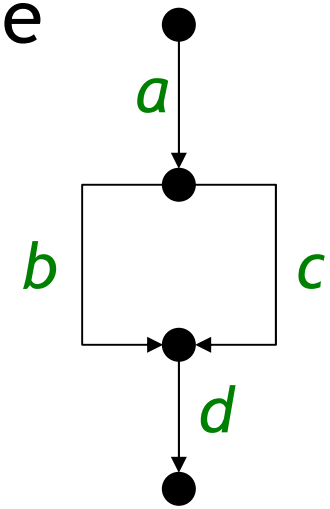
Sémantique formelle

- Un programme en AdP : terme algébrique

  $a \cdot (b + c) \cdot d$

  auquel on associe un modèle.

  La plupart du temps, il s'agit d'un STE (ex. pour CCS, CSP, LOTOS, LNT).



- Pour définir la sémantique d'une AdP, il suffit de définir la sémantique de chaque opérateur de l'AdP

Approche algébrique (sémantique axiomatique)

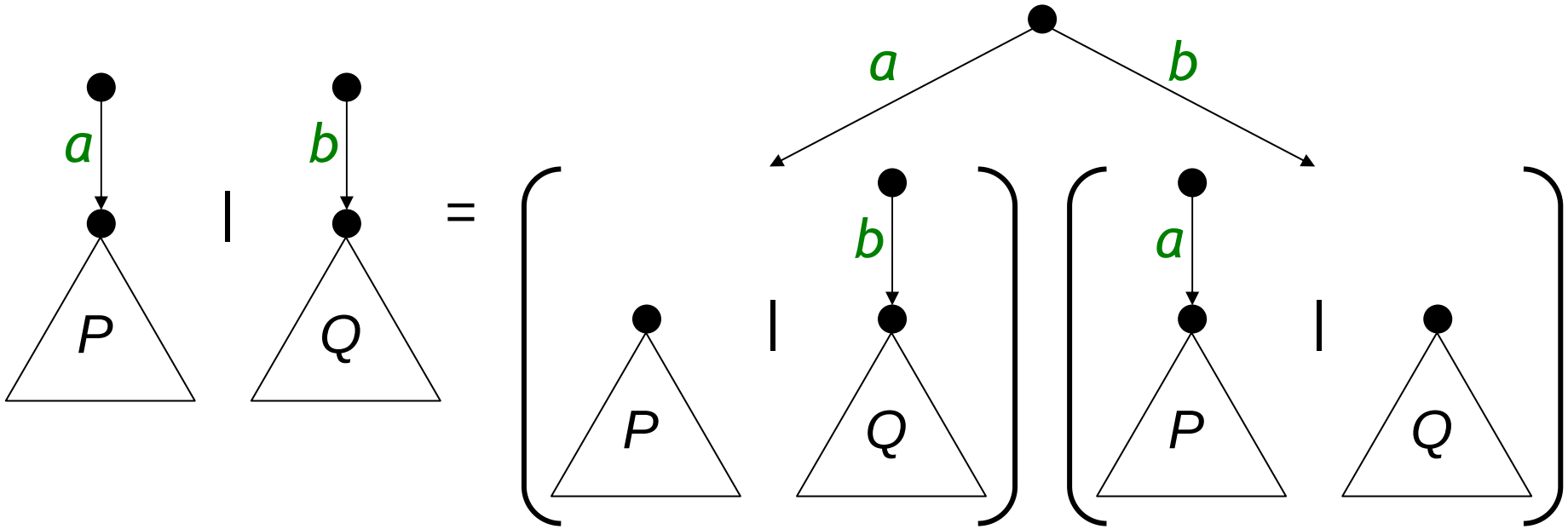
- Méthode utilisée pour définir la sémantique de CCS et ACP. Consiste à fournir un ensemble d'axiomes décrivant les relations entre opérateurs.
- Plusieurs axiomatisations peuvent être possibles $\Rightarrow$ problème de la *consistance* et de la *complétude*
- Ex. d'axiomes :

$B_1 + B_2 = B_2 + B_1$	commutativité de +
nil + $B = B$	**nil** élément neutre pour +
nil | $B = B$	**nil** élément neutre pour |

Théorème d'expansion

Le *théorème d'expansion* [Milner] permet de remplacer le parallélisme (' | ') par du choix (' + ') et de la séquence (' . ') :

$$a.P \mid b.Q = a.(P \mid b.Q) + b.(a.P \mid Q)$$



Entrelacement

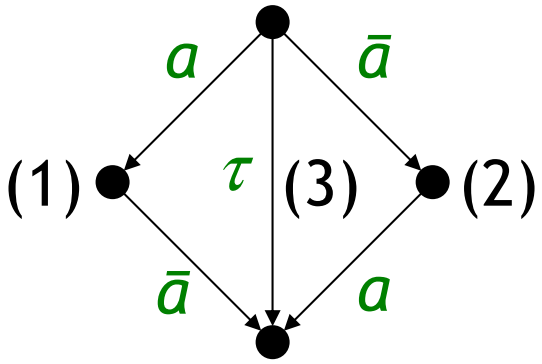
- Théorème d'expansion = base de la sémantique d'entrelacement (*interleaving semantics*)

$$a.\mathbf{nil} \mid b.\mathbf{nil} = a.b.\mathbf{nil} + b.a.\mathbf{nil}$$

Synchronisation en CCS

- En CCS : notion de « portes complémentaires »
- Exemple de synchronisation sur deux portes a et $\bar{a}$:

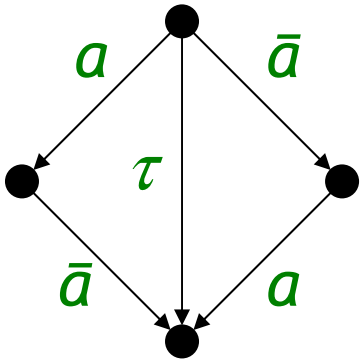
$$a.P \mid \bar{a}.Q = \begin{cases} a.(P \mid \bar{a}.Q) + & (1) \\ \bar{a}.(a.P \mid Q) + & (2) \\ \tau.(P \mid Q) & (3) \end{cases}$$



- L'opérateur '|' décrit le fait que les 2 processus peuvent (3) ou ne peuvent pas (1 et 2) se synchroniser. En pratique, on veut forcer la synchronisation ⇒ un autre opérateur (*restriction*) est nécessaire pour interdire (1) et (3) :

$$(a.P \mid \bar{a}.Q) \setminus a$$

Synchronisation en CCS (suite)



- Après synchronisation de a et $\bar{a}$, la transition devient cachée (renommée en τ) $\Rightarrow$ en CCS on peut modéliser uniquement du rendez-vous à deux
- Meilleures solutions en CSP, LOTOS, LNT : rendez-vous à *n*

Sémantique axiomatique (suite)

- La sémantique axiomatique permet de faire des *transformations* des programmes en appliquant les axiomes et le théorème d'expansion.

  On peut montrer la correction d'un programme parallèle en le transformant en un programme séquentiel plus simple [Milner].
- Inconvénient : en pratique, explosion combinatoire due à l'expansion du parallélisme
  $\Rightarrow$ il faut analyser des termes algébriques de grande taille

Sémantique opérationnelle

- Méthode utilisée pour définir la sémantique de LNT. Le comportement d'un terme algébrique (programme) est défini par un ensemble de règles de dérivation qui permettent de générer le STE correspondant.
- Ex. de règles (sémantique du choix) : op + de CCS

$$\frac{B_1 \xrightarrow{L} B_1'}{B_1 + B_2 \xrightarrow{L} B_1'} \qquad \frac{B_2 \xrightarrow{L} B_2'}{B_1 + B_2 \xrightarrow{L} B_2'}$$

Rem : cette approche n'a pas de problème de consistance ou complétude, à condition de respecter certaines conditions suffisantes sur le format des règles (SOS - *Structured Operational Semantics*).

Sémantique opérationnelle structurée

- Idée de SOS : éviter les règles sémantiques ayant des *prémisses négatives*

$$\frac{\neg\ (B_1 \xrightarrow{L} B_1')}{\ldots}$$

- Si prémisses négatives $\Rightarrow$ risque de divergence des règles (la construction du STE ne se termine pas). Dans ce cas, des conditions suffisantes plus fines sont nécessaires [Groote].
- Définitions SOS de CCS, CSP, LOTOS, LNT : pas de prémisses négatives

Prémisses négatives

Rem : il existe des cas où les prémisses négatives sont nécessaires

Exemple : exprimer *l'urgence* ou *la priorité* d'une action par rapport à une autre

$$\frac{B_1 \xrightarrow{L} B_1' \wedge \neg \left(B_2 \xrightarrow{M} B_2' \wedge prio\ (M) > prio\ (L) \right)}{B_1 + B_2 \xrightarrow{L}{}^{L}\ B_1'}$$

Traduction des termes vers des STE

Soit un terme B_0 d'une AdP. Comment construire le modèle STE correspondant ?

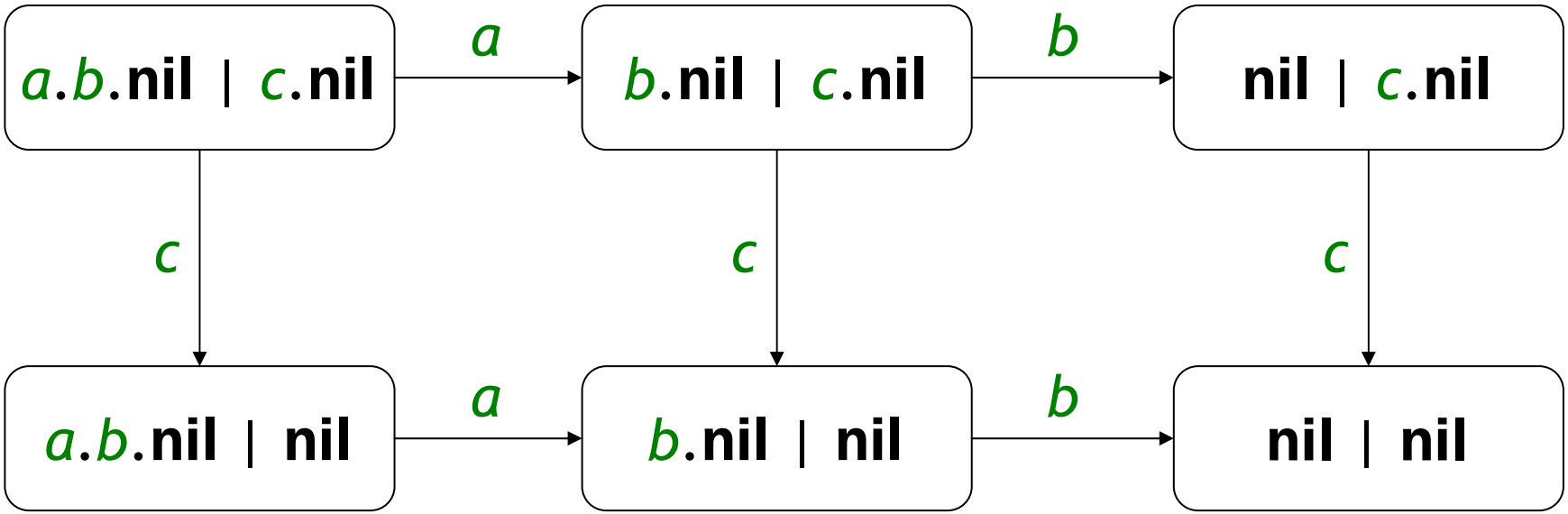
Associer à chaque terme un état du STE.

0. Ensemble initial d'états à explorer = { B_0 }.
1. S'il reste des états non-explorés, choisir un état B parmi ceux-ci.
2. Pour chaque règle applicable à B, qui exprime l'exécution d'une transition $B \xrightarrow{L} B'$ dans le STE, ajouter B' à l'ensemble (si pas déjà traité).
3. Rajouter la transition $B \xrightarrow{L} B'$ au STE et continuer en 1.

Exemple

Construction du STE du terme CCS :

(*a*.*b*.**nil** | *c*.**nil**)



Remarques

- Méthode automatique pour générer le modèle STE correspondant à une spécification algébrique
- Règles SOS bien définies $\Rightarrow$ terminaison et confluence de la méthode de construction
- Similaire à la construction de l'automate produit pour les AC ou du graphe des marquages pour les RdP
- Une fois le STE construit, le contenu des états n'est plus nécessaire (le comportement du système est défini par les *actions* du STE)
- On peut appliquer sur le STE les techniques de vérification par model-checking

Exercice

- Etant données les règles SOS simplifiées de CCS ci-dessous (sans synchro), dessiner le STE du terme

  a.(d.**nil** + e.**nil**) + b.c.(d.**nil** | e.**nil**)

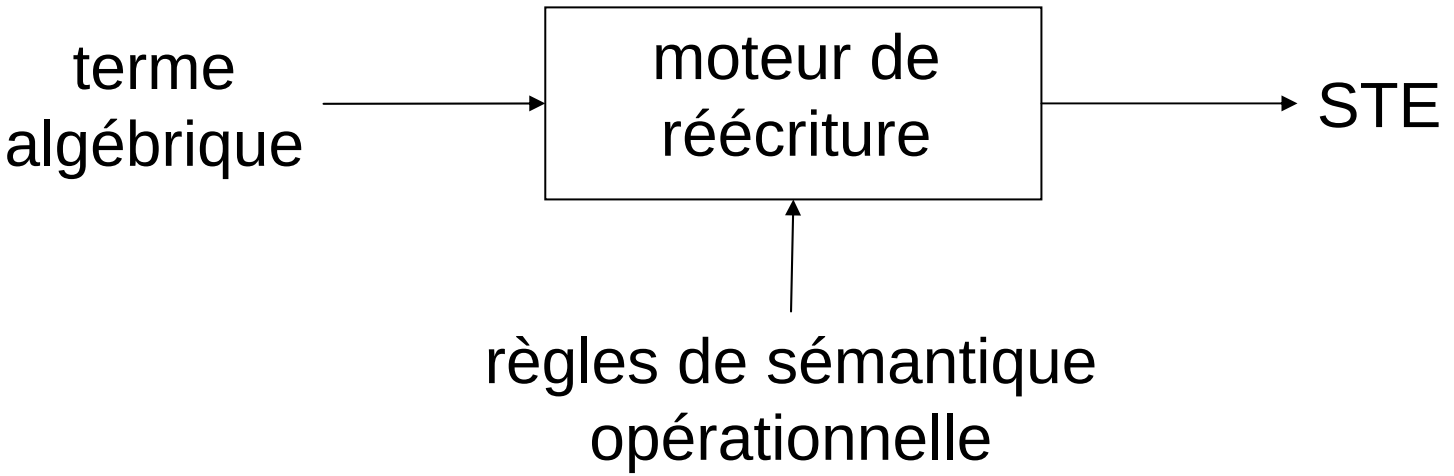
$$\frac{}{a.B \xrightarrow{a} B} \qquad \frac{B_1 \xrightarrow{a} B_1'}{B_1 + B_2 \xrightarrow{a} B_1'} \qquad \frac{B_2 \xrightarrow{a} B_2'}{B_1 + B_2 \xrightarrow{a} B_2'}$$

$$\frac{B_1 \xrightarrow{a} B_1'}{B_1 \mid B_2 \xrightarrow{a} B_1' \mid B_2} \qquad \frac{B_2 \xrightarrow{a} B_2'}{B_1 \mid B_2 \xrightarrow{a} B_1 \mid B_2'}$$

Implémentation

1ère solution : réécriture de termes

- Les règles SOS sont données comme entrée à un moteur (général) de réécriture

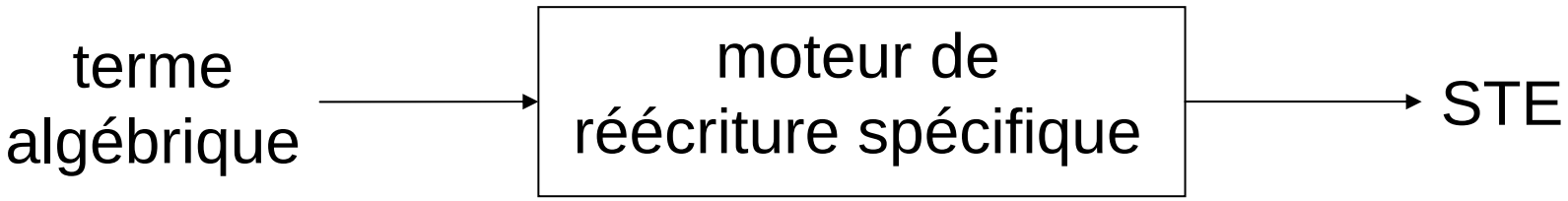


- Peu efficace (manipulation de gros termes) mais automatisable. PAC : *Process Algebra Compiler*

Implémentation (suite)

2ème solution : réécriture de termes optimisée

- Les règles SOS sont implantées dans un moteur de réécriture spécifique



- Plus efficace en pratique ; approche utilisée dans des outils pour CCS (Concurrency Workbench) et LOTOS (HIPPO, SMILE)

Implémentation (suite)

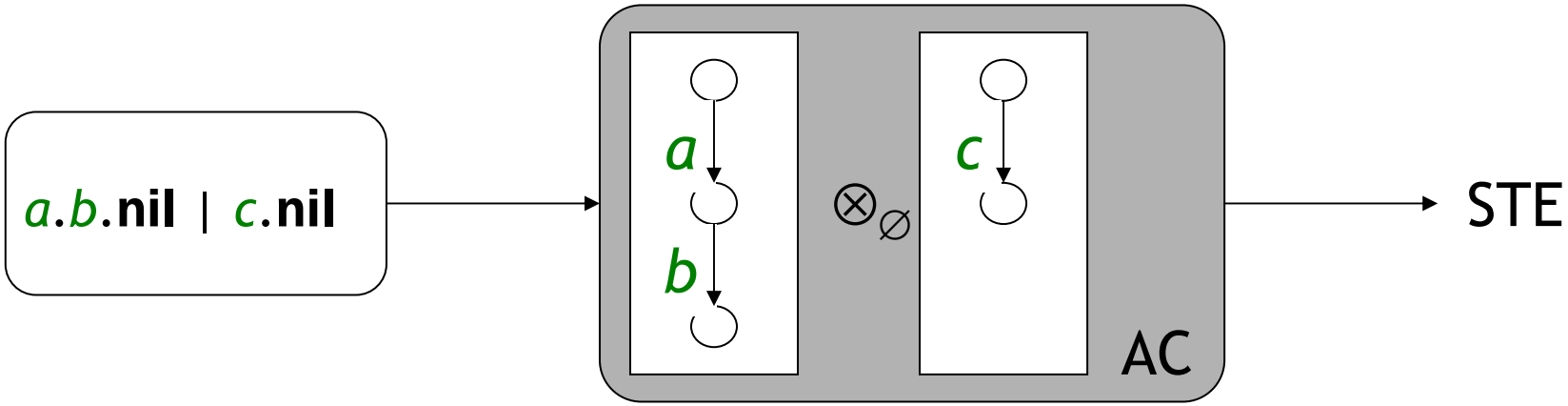
Les solutions basées sur la réécriture de termes ont deux sources d'inefficacité :

- **Consommation mémoire importante**
  - un état du STE : terme algébrique (= arbre abstrait d'expression)
  - état initial = arbre du programme source
  - états suivants = arbres expansés
- **Vitesse réduite**
  - exécution lente des transitions (réécriture)
  - comparaison lente des états (parcours d'arbres)

Implémentation (suite)

3ème solution : traduction du terme vers un système d'AC, puis construction de l'automate produit

Exemple :

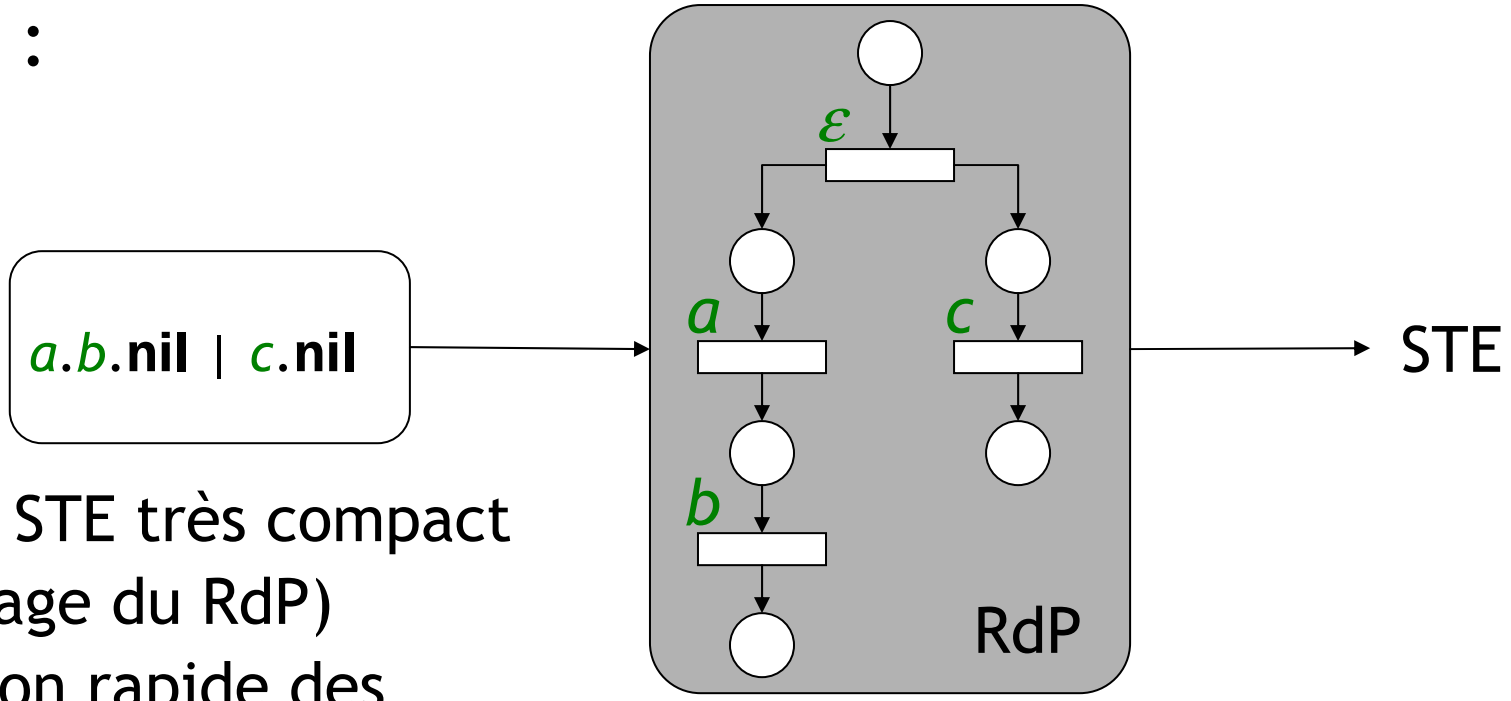


- état du STE très compact (liste de numéros d'états des AC)
- exécution rapide des transitions (produit synchrone)
- mais solution moins générale

Implémentation (suite)

4ème solution : traduction du terme vers un RdP, puis construction du graphe des marquages

Exemple :



- état du STE très compact (marquage du RdP)
- exécution rapide des transitions (calcul du marquage successeur)
- plus général que les automates communicants

LOTOS NT (LNT)

Héritier de LOTOS (norme internationale [ISO 8807] pour la spécification formelle des protocoles de télécommunication et des systèmes distribués)

Variante de la norme E-LOTOS [ISO 15437]

Combinaison des concepts AdP avec un langage algorithmique classique (impératif / fonctionnel)

- partie *données* (pour le calcul sur les structures de données) : types, fonctions, instructions
- partie *contrôle* = sur-ensemble de la partie données : comportements

Rem : une partie données est nécessaire pour décrire des systèmes réalistes. Il existe des sous-ensembles des AdP sans données (« pure CCS », « basic LOTOS »), mais qui sont inutilisables en pratique.

LNT

- LNT supporté par la boîte à outils CADP (INRIA / VASY) : simulation, vérification formelle (*model checking*), ...
- Traduction vers LOTOS (lnt2lotos, lnt.open)
- Dans ce cours : présentation partielle de LNT et de sa sémantique formelle
- Documentation complète [doc] : voir Kiosk ou
  $CADP/doc/pdf/Champelovier-Clerc-Garavel-et-al-10.pdf
- **Rem** : CADP est installé sur ensisun
  Pour l'utiliser, définir les variables suivantes :

```
export CADP=/usr/local/cadp
PATH="$CADP/com:$CADP/bin.x64:$PATH"
```

Formalisme de présentation du langage

- Grammaire EBNF (*Extended Bachus Naur Form*)
  - Mots-clés et symboles terminaux en **rouge**
  - Meta-symboles en **bleu**
  - Symboles non-terminaux dans les autres couleurs
- Règles de la forme Θ ::= E_0 | ... | E_n où :
  - Θ : symbole non-terminal
  - E_0, ..., E_n : séquences de symboles (terminaux et non-terminaux)
  - [E] : production optionnelle
  - E_0 ... E_n : répétition non-vide (E+)
  - E_1 ... E_n : répétition possiblement vide (E*)

Définitions de types de données

- Identificateurs de types T, T_1, T_2, ...
  - Prédéfinis : bool, char, nat, int, real, string
  - Définis par l'utilisateur : **type** T **is** E **end type**
- Expressions de types E, E_1, E_2, ...

E ::= D_0, ..., D_n

| **set of** T | **list of** T |

| **array [** n **..** m **] of** T | ...

D ::= C [(X_1 : T_1, ..., X_n : T_n)]

D_i : définition de constructeurs

X_i : variable

C : identificateur de constructeur

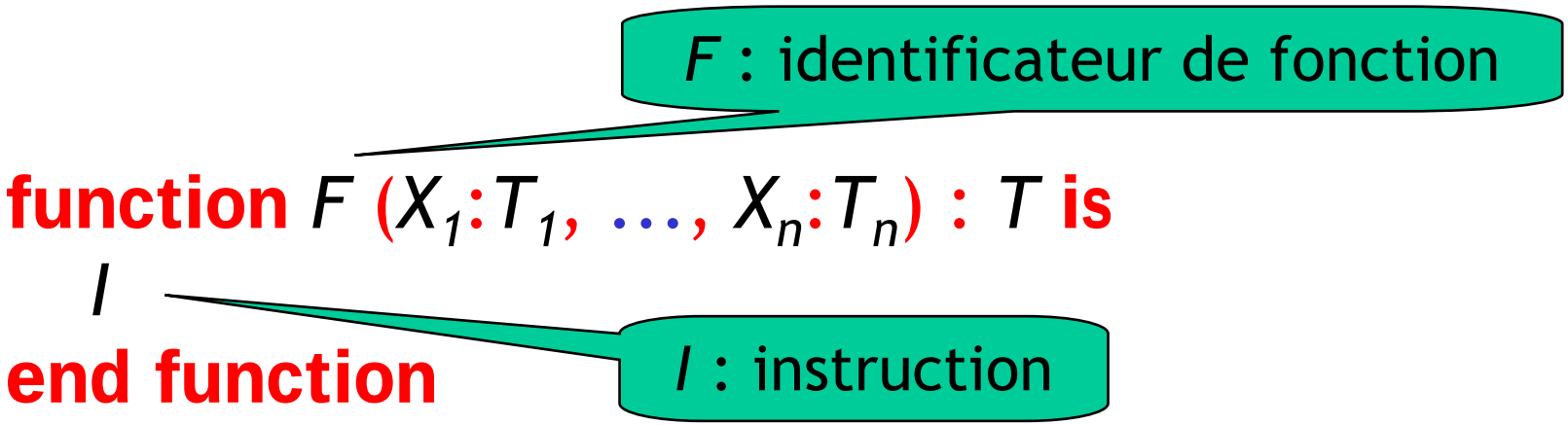
Types à constructeurs

- Issus des langages fonctionnels (ML, Haskell)
- **Exemples :**
  - **Enuméré :**
    type color **is** red, green, blue **end type**
  - **Record :**
    type pers **is** pair (name : string, age : nat) **end type**
  - **Union :**
    type IntBool **is** anInt (n : int), aBool (b : bool) **end type**
  - **Liste :**
    type int_list **is** nil, cons (hd : int, tl : int_list) **end type**
    équivalent à
    type int_list **is list of** int **end type**
  - **Arbre binaire :**
    type tree **is** leaf, node (fg : tree, fd : tree) **end type**

Complément sur les types de données

- Possibilité de types externes en C
- Définition (optionnelle) d'opérations par défaut pour certains types (accesseurs, comparaisons, ...)
- Addition récente des types intervalle et sous-types à prédicats
- etc., voir [doc]

Définitions de fonctions



- Ici : passage de paramètres par valeur (autres modes de passage permis, voir plus loin)
- Effet défini par une *instruction* : (affectations, séquence, conditionnelles, boucles, return, ...)

Exemple

```
function fact (n : nat) : nat is
   var result : nat in
      result := 1;
      while n > 1 loop
            result := result * n; n := n - 1
      end loop;
      return result
   end var
end function
```

; séparateur (≠ C terminateur)

Exemples d'appel :

- fact (4 of nat)
- fact (X)

Passage de paramètres par référence (paramètres out et inout)

- Permis aussi par LNT mais non-présenté dans la grammaire ci-dessus ; voir [doc]
- **Exemple** (incrémentation compteur 16 bits avec vérification d'overflow) :

```
function incr16 (inout n : nat, in delta : nat,
                 out overflow : bool) is
  if n >= 65535 - delta then overflow := true
  else overflow := false; n := n + delta end if
end function
```

Expressions

- Notées V, V_1, V_2, ...
- Expression = terme qui a une valeur unique dans un contexte donné

V ::= X

| C [(V_1, ..., V_n)]

| F [(V_1, ..., V_n)]

| V[X]

| V.X | V.{X_0=>V_0, ..., X_n=>V_n}

| (V_0) | V_0 **of** T | ...

C et *F* existent aussi en notation infixe, voir [doc]

Coercion de type

Surcharge de symboles

- Possibilité d'avoir des fonctions ou constructeurs de même nom mais de types différents

  function odd (n : nat) : bool **is** ... **end function**

  function odd (n : int) : bool **is** ... **end function**

- Idem pour les fonctions prédéfinies

  function _+_ (b1 : Bool, b2 : Bool) : Bool **is**

  ...

  end function

  opérateur infixe

- **Attention** : pas de règles de priorité entre opérateurs ⇒ toujours parenthéser !

  a and b or c **doit être écrit** (a and b) or c

Motifs (*patterns*)

- Notés P, P_1, P_2, ...
- Motif = terme qui représente un ensemble de valeurs possibles dans un contexte donné

  P ::= X

  | **any** T

  | C [(P_0, ..., P_n)]

  | P_0 **where** V_0 | ...

- Intérêt : *filtrage de motifs* (*pattern-matching*) hérité des langages fonctionnels (ML, Haskell, ...)

Existe aussi en notation infixe, voir [doc]

Filtrage de motif

- Instruction **case**

  case *V* **in** [**var** X_1:T_1, …, X_m:T_m] **in**
  P_0 -> I_0 | … | P_n -> I_n
  end case

- Si *V* appartient à l'ensemble de valeurs représentées par P_i ($i \in$ 1..n) alors exécuter I_i (avec priorité de gauche à droite)
- Les variables de P_i prennent la valeur des sous-termes correspondant dans *V*

Exemple de filtrage de motif

```
type arbre is
    f,
    n (fg : arbre, fd : arbre)
end type
```

feuille

noeud binaire

taille

```
function t (a : arbre) : nat is
    case a in
    var fg : arbre, fd : arbre in
       f -> return 1
    | n (fg, fd) -> return t (fg) + t (fd) + 1
    end case
end function
```

Exemple : a = n (f, f)
résultat:
t (f) + t (f) + 1 = 3

Partie contrôle

Permet la définition des processus (comportements B, B_1, B_2, ...)

Particularité de LNT : symétrie entre données et contrôle

- Les comportements englobent toutes les instructions de la partie donnée
- Exception : **return** (réservé à la partie données)

Sont ajoutés des comportements décrivant :

- le non-déterminisme
- le parallélisme asynchrone
- la communication
- l'abstraction de comportement

interdits dans la partie données !

Comportements existant dans la partie données (1/2)

B ::= **null**

| B_1; B_2

| X := V

| X[V_0] := V_1

| **var** X_1:T_1, …, X_n:T_n **in** B_0 **end var**

| **case** V **in** [**var** X_1:T_1, …, X_m:T_m] **in**

P_0 -> B_0 | … | P_n -> B_n

end case

| …

Comportements existant dans la partie données (2/2)

B ::= …

| **if** V_0 **then** B_0
  [**elsif** V_1 **then** B_1 … **elsif** V_n **then** B_n]
  [**else** B_{n+1}] **end if**

| **loop** [*L* **in**] B_0 **end loop**

| **break** *L*

| **while** *V* **loop** B_0 **end loop**

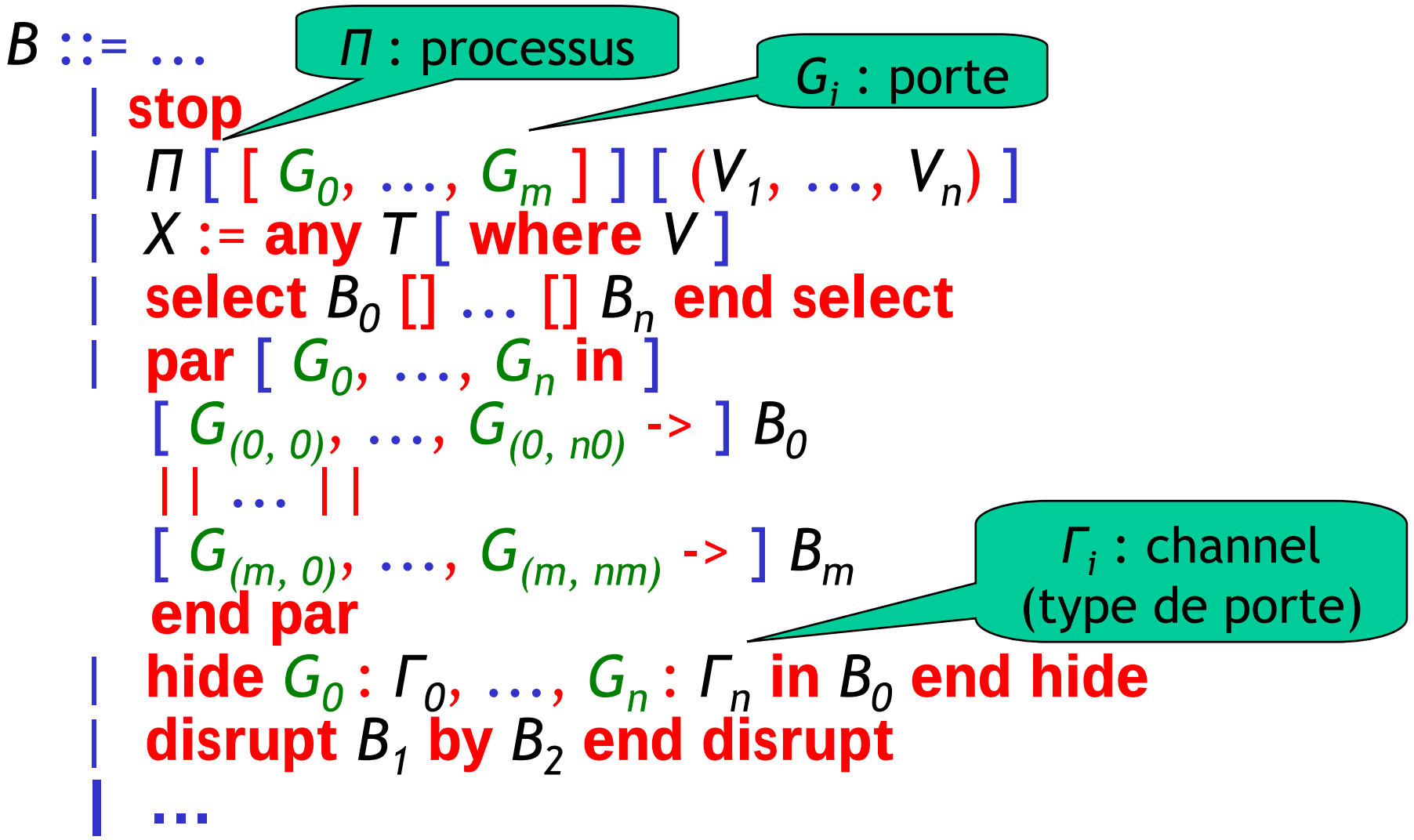
| **for** B_0 **while** *V* **by** B_1 **loop** B_2 **end loop**

| …

L : identificateur de boucle

+ appels de procédures et exceptions, voir [doc]

Comportements spécifiques à la partie contrôle (1/2)



Comportements spécifiques à la partie contrôle (2/2)

Communication

B ::= ...
| *G* [(*O*$_1$, ..., *O*$_n$)] [**where** *V*]

Offres de communication *O*, *O*$_1$, *O*$_2$, ...

O ::= [**!**] *V*
| **?** *P*

émission de la valeur de *V*

réception d'une valeur filtrée par le motif *P*

Channels

- Possibilité, au moment de déclarer une porte, de contraindre le type des données échangées sur cette porte
- Identificateurs de channels Γ, Γ_1, Γ_2, ...
  - Prédéfini : **any** (aucune contrainte)
  - Définis par l'utilisateur

    channel Γ **is** K_0, ..., K_n **end channel**

    avec K ::= (T_1, ..., T_n)

    permet le transit de tuples dont les valeurs ont les types respectifs T_1, ..., T_n

Exemples

Variables : *N* : nat, *B* : bool

- **channel** ch1 **is** (nat) **end channel**

  G (4), *G* (?*N*), etc.

- **channel** ch2 **is** (nat), (bool) **end channel**

  G (4), *G* (?*N*), *G* (true), *G* (?*B*), etc.

- **channel** ch3 **is** (), (nat, bool) **end channel**

  G, *G* (4, ?*B*), *G* (?*N*, ?*B*), *G* (4, true), etc.

Définitions de processus

process Π [[G_0:Γ_0, …, G_n:Γ_n]] [(X_1:T_1, …, X_n:T_n)]
is B **end process**

appel par valeur, voir [doc]
pour l'appel par référence

où :

- Π = nom du processus
- G_0, …, G_n = paramètres formels *portes* de Π
- Γ_0, …, Γ_n = channels de G_0, …, G_n
- X_1, …, X_n = paramètres formels *données* de Π
- T_1, …, T_n = types de X_1, …, X_n
- B = comportement de Π

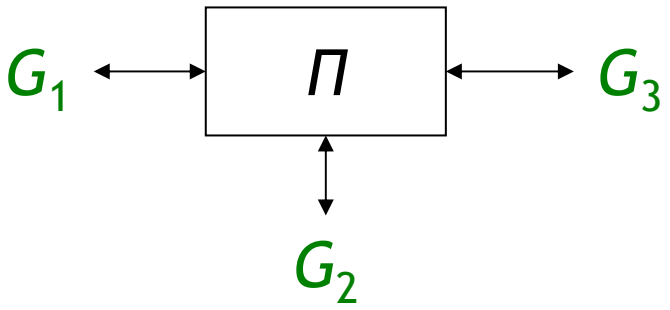
Exemple

```
channel none is () end channel

process   SEMAPHORE   [REL : none, REQ : none]
                      (locked : bool) is
   ... (* description du comportement *)
end process
```

Remarques

- Un processus LNT est une « boîte noire » avec des points de communication sur l'extérieur



process Π [G_1:Γ_1, G_2:Γ_2, G_3:Γ_3]
(...) **is**

...

end process

- On peut décrire des mises en parallèle de boîtes au moyen des opérateurs **par** et **hide**

Mise en parallèle (1/2)

par [G_0, …, G_n **in**]
  [$G_{(0,0)}$, …, $G_{(0,n0)}$ ->] B_0
  || … ||
  [$G_{(m,0)}$, …, $G_{(m,nm)}$ ->] B_m
end par

Généralisation du $B_1 \otimes \ldots \otimes B_n$ des AC

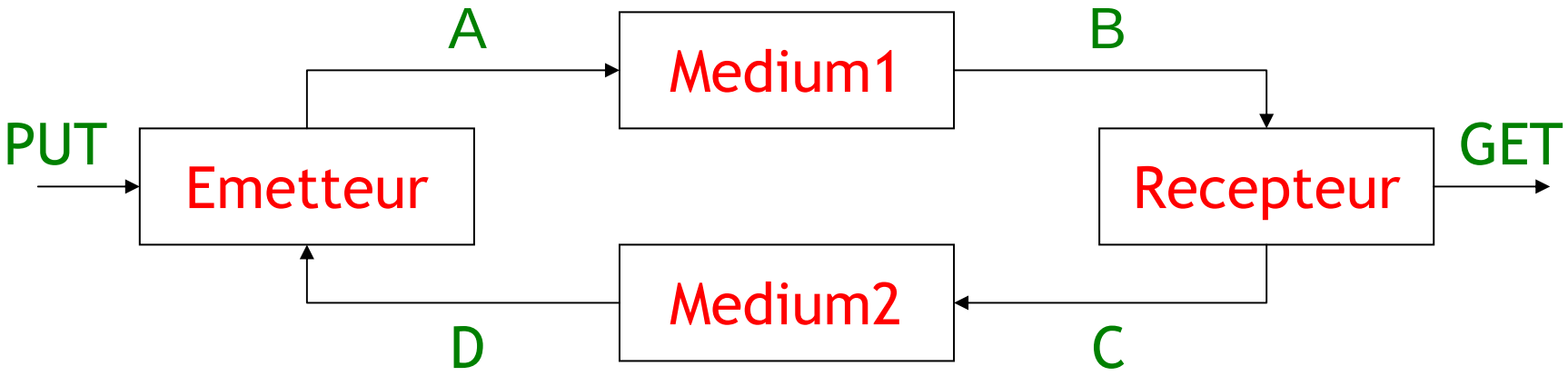
B_0, …, B_m s'exécutent en parallèle, avec :

- obligation pour tous de se synchroniser sur G_0, …, G_n
- pour toute porte G, obligation pour tous les B_i tels que $G_{(i,0)}$, …, $G_{(i,ni)}$ contient G de se synchroniser sur G
- interdiction de se synchroniser sur les autres portes

Mise en parallèle (2/2)

Remarque : si B_0, ..., B_m se terminent, ils doivent le faire en même temps (*join*). Dans la sémantique, ceci est modélisé par une synchronisation sur une porte spéciale **exit** qui n'apparaît dans aucun programme LNT.

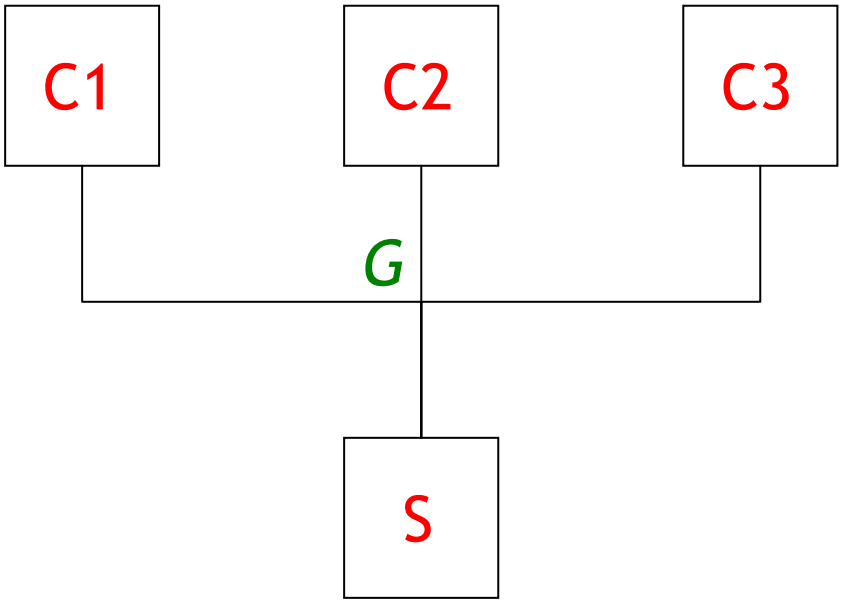
Exemple




```
par
        A, D -> Emetteur [PUT, A, D]
||      B, C -> Recepteur [GET, B, C]
||      A, B -> Medium1 [A, B]
||      C, D -> Medium2 [C, D]
end par
```

Mise en parallèle (suite)

Les opérateurs de LNT permettent d'exprimer la synchronisation de $n \geq 2$ processus sur la même porte (plus puissant que CCS).



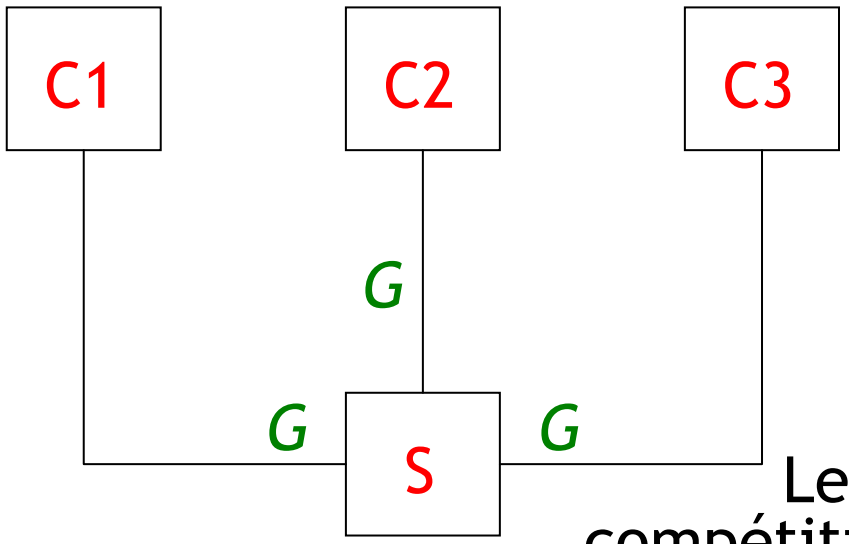
Exemple (client-serveur) :

```
par G in
C1 [G] || C2 [G] || C3 [G] || S [G]
end par
```

Les trois processus clients se synchronisent sur *G* avec le processus serveur.

Mise en parallèle (suite)

L'opérateur **par** permet de modéliser des connexions binaires (synchronisation à 2) sur une même porte.



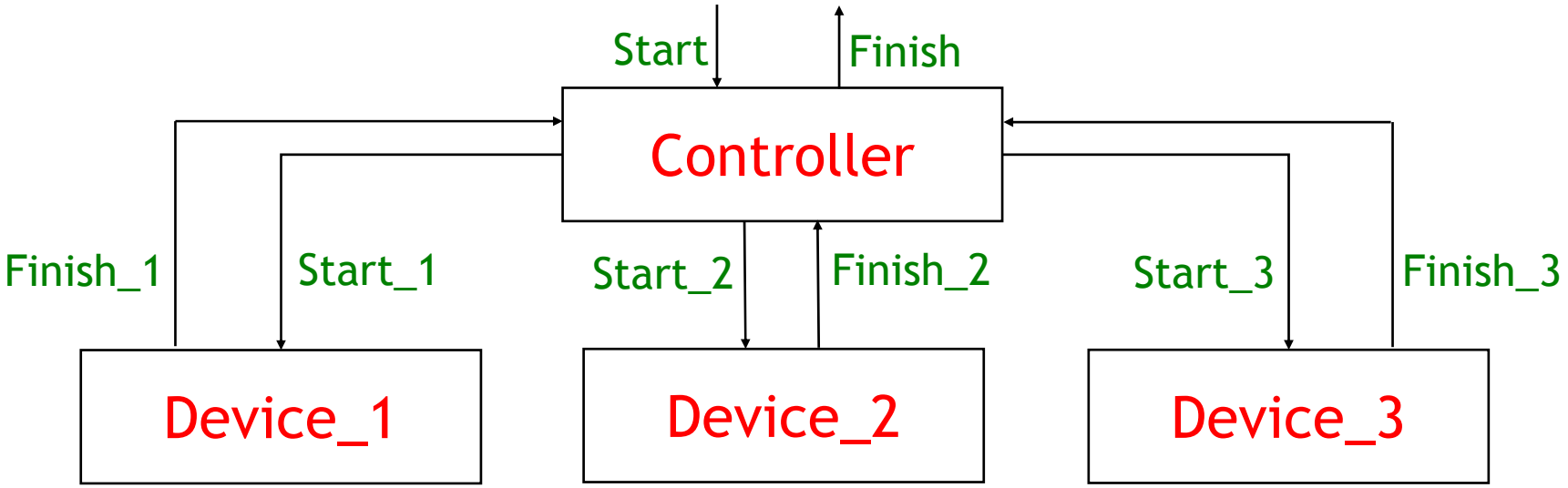
Exemple (client-serveur) :

```
par G in
        par
        C1 [G] || C2 [G] || C3 [G]
        end par
|| S [G]
end par
```

Les trois clients sont en compétition pour accéder au serveur (synchronisation à deux sur *G*). Si plusieurs clients veulent exécuter *G* ⇒ choix non-déterministe.

Exercice

- Ecrire une expression de composition parallèle correspondant à l'architecture parallèle suivante :



Remarque

- Actuellement, les outils de CADP n'autorisent que les **par** qui peuvent se traduire en n'utilisant que des comportements de la forme

  par G_1, ..., G_n **in** B_0 || ... || B_m **end par**

  (c'est souvent le cas en pratique)
- **Exemple** :

```
par
    A, D -> Emetteur [PUT, A, D]
|| B, C -> Recepteur [GET, B, C]
|| A, B -> Medium1 [A, B]
|| C, D -> Medium2 [C, D]
end par
```

```
par A, B, C, D in
    par
        Emetteur [PUT, A, D]
    || Recepteur [GET, B, C]
    end par
|| par
        Medium1 [A, B]
    || Medium2 [C, D]
    end par
end par
```

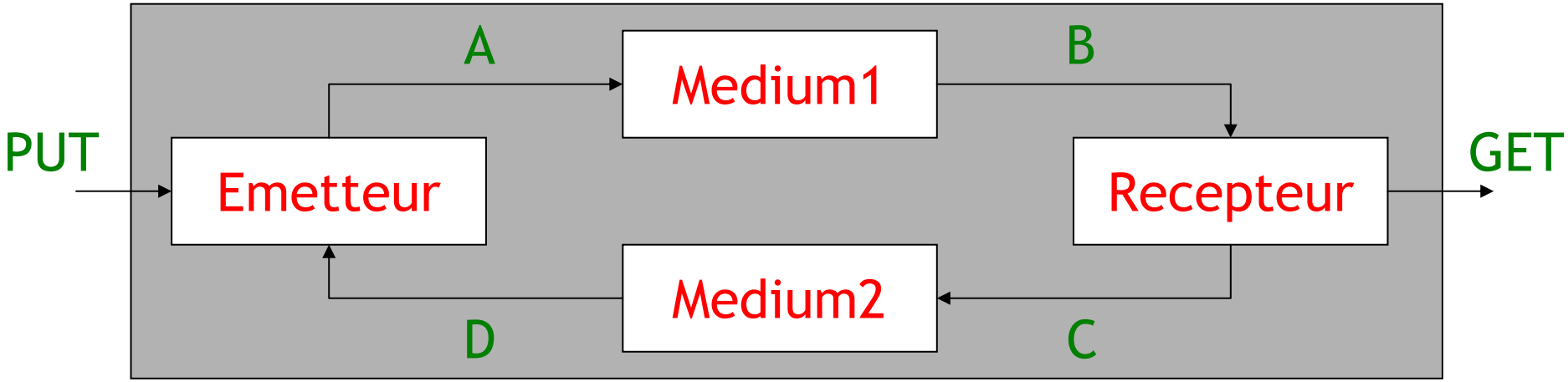
Abstraction (opérateur hide)

- En LNT (comme en LOTOS), au moment de la synchronisation, les portes ne sont pas renommées en τ ($\neq$ CCS) $\Rightarrow$ d'autres processus peuvent participer à la synchronisation
- Si on veut éviter ceci, on peut cacher les portes (les renommer en τ) au moyen de l'opérateur **hide**.

  hide $G_0 : \Gamma_1, \ldots, G_n : \Gamma_n$ **in** B **end hide**

  signifie qu'à l'intérieur de B, toutes les occurrences de $G_0, \ldots, G_n$ sont renommées en τ
- On dit que les portes $G_0, \ldots, G_n$ sont « abstraites » (ignorées depuis l'extérieur)

Exemple




```
channel msg is
    (nat)
end channel

channel none is
    ()
end channel
```

```
process Reseau [PUT : msg, GET : msg] is
  hide A : msg, B : msg, C : none, D : none in par
        A, D -> Emetteur [PUT, A, D]
    ||  B, C -> Recepteur [GET, B, C]
    ||  A, B -> Medium1 [A, B]
    ||  C, D -> Medium2 [C, D]
  end par end hide
end process
```

Sémantique statique (1/2)

Garantit la **bonne formation** des programmes

- **Liaison :** toute variable doit être déclarée (paramètre formel ou bloc **var** ... **end var**)
- **Typage :** même type = même nom de type
- **Initialisation :** toute variable doit avoir été définie avant sa première utilisation (*cf.* Java)
- Une variable définie / modifiée dans une branche ne doit pas être utilisée dans les branches parallèles

  par G (?X) || **case** Y **in** C (X) -> ... **end par** 

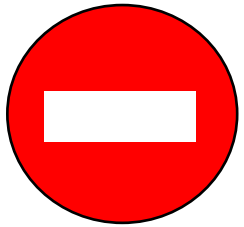
  par X := 2 || **if** X = 0 **then** ... **end par** 

Sémantique statique (2/2)

Restrictions concernant les processus récursifs

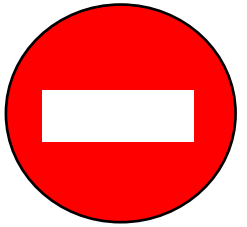
- Récursion **terminale** seulement

```
process P [G : none] is
    ... P[G]; ...
end process
```



- Pas de récursion à travers le **par** (réplication)

```
process P [G : none] is
    par ... || ... P [G] ... || ... end par
end process
```



Règles de sémantique opérationnelle

Notations utilisées :

- v : **valeur** = expression sans variables ni symboles de fonctions
- ρ : **contexte** = fonction partielle des variables vers les valeurs
  (les tableaux sont omis par simplicité)
  notations : $\rho = [\ X_1 := v_1,\ \ldots,\ X_n := v_n\ ]$
  $\mathrm{dom}(\rho) = \{X_1,\ \ldots,\ X_n\}$

Opérations sur les contextes

- **Extension** de contexte :

  ρ etendu par ρ'

$$(\rho + \rho')\ (X) = \begin{cases} \rho'(X) & \text{si } X \in \text{dom}(\rho') \\ \rho(X) & \text{si } X \in \text{dom}(\rho) \setminus \text{dom}(\rho') \\ \text{non-défini} & \text{sinon} \end{cases}$$

- **Différence** de contextes :

  ce qui a changé de ρ à ρ'

$$(\rho' - \rho)\ (X) = \begin{cases} \rho'(X) & \text{si } X \in \text{dom}(\rho') \text{ et} \\ & (X \in \text{dom}(\rho) \Rightarrow \rho'(X) \neq \rho(X)) \\ \text{non-défini} & \text{sinon} \end{cases}$$

Sémantique des expressions

- Relation de la forme $\{ V \}\ \rho \rightarrow_e v$
- Dans le contexte ρ, V s'évalue en v
- Voir [doc] pour la définition formelle (classique et intuitive)
- **Exemple**

$$\{ X + Y \}\ [X := 1, Y := 4] \rightarrow_e 5$$

Sémantique du filtrage de motifs

- Relation de la forme
  - { *P* # v } $\rho \rightarrow_p \rho'$: dans le contexte ρ, *P* filtre la valeur *v* et produit le contexte ρ'

  ou
  - { *P* # v } $\rho \rightarrow_p$ **fail** : dans le contexte ρ, *P* ne filtre pas la valeur *v*
- Voir [doc] pour la définition formelle
- **Exemples**

  { pers (n, a) # pers ("Jo", 32) } [x := 1, a := 0] $\rightarrow_p$ [x := 1, n := "Jo", a := 32]

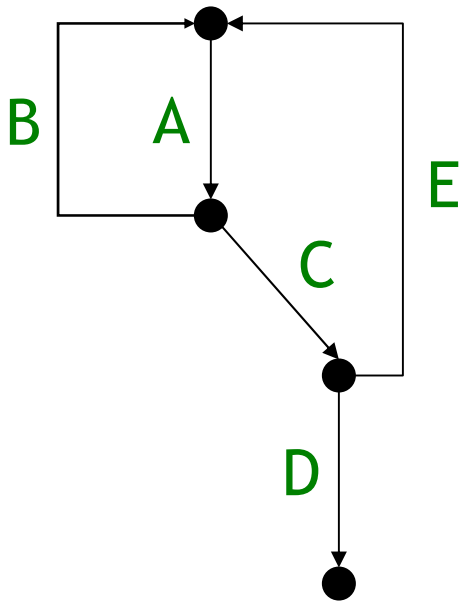
  { node (fg, fd) # leaf } [] $\rightarrow_p$ **fail**

Sémantique des comportements

- Relation de la forme $\{ B \}\ \rho\ -l\rightarrow_b \{ B' \}\ \rho'$: dans le contexte ρ, B produit le contexte ρ' et doit continuer à s'exécuter comme B' et
  - l = **exit** : B se termine normalement
  - $l = G\ (v_1, \ldots, v_n)$: B fait une communication
- La sémantique d'un comportement LNT principal B_0 est un LTS
  - état initial $\{ B_0 \}$ []
  - transitions définies par la relation $\{ B \}\ \rho\ -l\rightarrow_b \{ B' \}\ \rho'$
- **Rem :** Le label **exit** mène toujours à un état de deadlock (l = **exit** $\Rightarrow$ B' = **stop**)

Comportements séquentiels

LNT permet de coder les comportements séquentiels au moyen d'opérateurs de choix (**select**), séquencement (**stop** et ;), boucles (loop) et/ou processus récursif terminal




```
process P [A, B, C, D, E : any] is
   loop
      A;
      select
         B
      []
         C;
         select
            D; stop
         []
            E
         end select
      end select
   end loop
end process
```

style itératif

ou

```
process P [A, B, C, D, E : any] is
   A;
   select
      B;
      P [A, B, C, D, E]
   []
      C;
      select
         D; stop
      []
         E;
         P [A, B, C, D, E]
      end select
   end select
end process
```

style récursif

Sémantique de « stop »

Pour l'opérateur **stop** (inaction) : aucune règle sémantique associée. On ne peut dériver aucune action à partir de **stop**.

L'opérateur **stop** n'existe pas dans la partie données

Sémantique de « null »

- Inaction : **null** se termine immédiatement sans changer le contexte

$$\frac{}{\{\ \textbf{null}\ \}\ \rho\ \xrightarrow{\textbf{exit}}_{b}\ \{\ \textbf{stop}\ \}\ \rho}$$

Communication sur une porte (1/2)

B ::= ... | G [(O_1, ..., O_n)] [**where** V]

O ::= [**!**] V | **?** P

(on peut mélanger librement des offres d'émission et de réception dans la même communication)

Sémantique des offres :

Relation de la forme $\{ O \# v \} \rho \rightarrow_o \rho'$:

dans le contexte ρ, O accepte la valeur v et produit le contexte ρ'

$$\frac{\{ V \} \rho \rightarrow_e v}{\{ \mathbf{!}\ V \# v \} \rho \rightarrow_o \rho} \qquad \frac{\{ P \# v \} \rho \rightarrow_p \rho'}{\{ \mathbf{?}\ P \# v \} \rho \rightarrow_o \rho'}$$

Communication sur une porte (2/2)

$$\rho_0 = \rho \qquad (\forall i \in 1..n)\ \{\ O_i \ \#\ v_i\ \}\ \rho_{i-1} \rightarrow_o \rho_i \qquad \rho' = \rho_n$$
$$\{\ V\ \}\ \rho' \rightarrow_e \text{true}$$

$$\{\ G\ (O_1, \ldots, O_n)\ \textbf{where}\ V\ \}\ \rho\ -G\ (v_1, \ldots, v_n)\rightarrow_b \{\ \textbf{null}\ \}\ \rho'$$

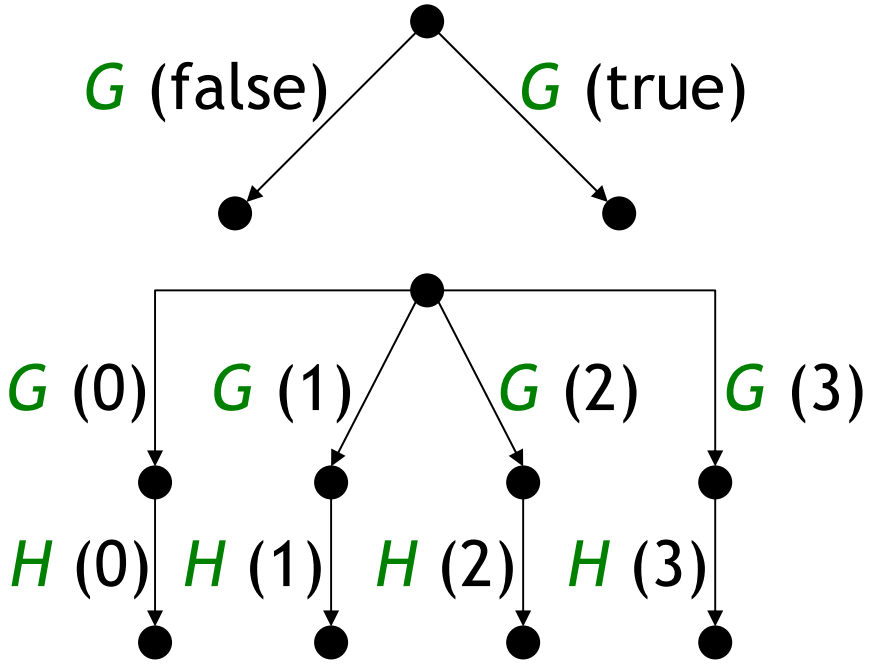
Rem :

- n'existe pas dans la partie données
- après, terminaison normale (**null**)
- si la garde *V* est fausse, le RdV n'a pas lieu (blocage) : $G\ (O_1, \ldots, O_n)$ **where** false ≈ **stop**
- si plusieurs valeurs satisfont les conditions, il y a un choix non-déterministe entre ces valeurs

Exemples

G (?*X*); **stop**
avec *X*:bool

G (?*X*) **where** *X* < 4;
H (*X*); **stop**
avec *X*:nat

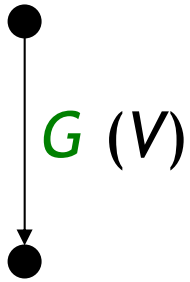


Rem : la sémantique traite la réception en itérant sur toutes les valeurs qu'on peut recevoir. Les gardes permettent de limiter le domaine des variables de réception.

Exemples (suite)

- Emission d'une valeur = réception gardée :

$G\ (V) \equiv G\ (?X)$ **where** $X = V$



Sémantique de « ; »

$$B_1 ; B_2$$

- **Communication dans B_1**

$$\frac{\{ B_1 \}\ \rho \xrightarrow{G\ (v_1, \ldots, v_n)}_{\mathrm{b}} \{ B_1' \}\ \rho'}{\{ B_1 ; B_2 \}\ \rho \xrightarrow{G\ (v_1, \ldots, v_n)}_{\mathrm{b}} \{ B_1' ; B_2 \}\ \rho'}$$

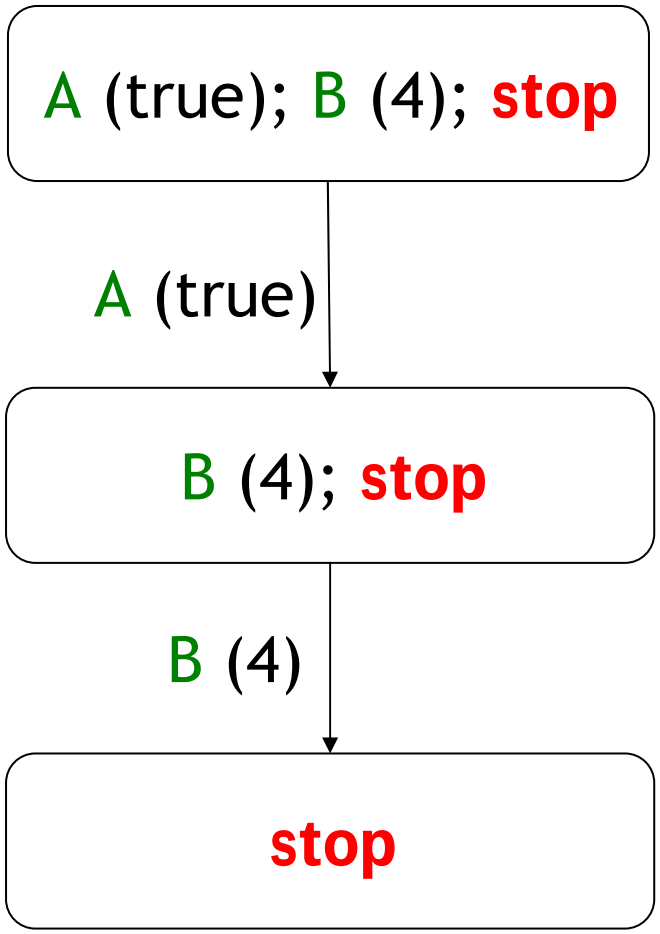
- **Terminaison de B_1**

$$\frac{\{ B_1 \}\ \rho \xrightarrow{\mathbf{exit}}_{\mathrm{b}} \{ B_1' \}\ \rho' \qquad \{ B_2 \}\ \rho' \xrightarrow{l}_{\mathrm{b}} \{ B_2' \}\ \rho''}{\{ B_1 ; B_2 \}\ \rho \xrightarrow{l}_{\mathrm{b}} \{ B_2' \}\ \rho''}$$

Exemples

Composition séquentielle :

A (true); B (4); **stop**



Sémantique de « case »

$$\frac{\{ V \}\ \rho \rightarrow_e v \qquad j \in 1..m \qquad (\forall i \in 1..j\text{-}1)\ \{ P_i \# v \}\ \rho \rightarrow_p \textbf{fail} \qquad \{P_j \# v\} \rightarrow_p \rho_j \qquad \{ B_j \}\ \rho_j \text{ -}l\rightarrow_b \{ B_j' \}\ \rho_j'}{\{ \textbf{case } V \textbf{ in } P_1 \text{ -> } B_1 \mid \ldots \mid P_m \text{ -> } B_m \textbf{ end case} \}\ \rho \text{ -}l\rightarrow_b \{ B_j' \}\ \rho_j'}$$

Rem : un comportement **if-then-else** est expansé vers un comportement **case**

if V **then** B_1 **else** B_2 **end if**

≡ **case** V **in** true -> B_1 | false -> B_2 **end case**

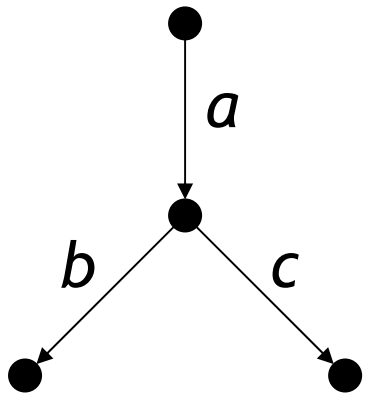
Opérateur « select »

- Exprime le choix *non-déterministe* :

  select B_1 [] ... [] B_n **end select**

  on peut exécuter soit B_1, ..., soit B_n

```
a ;
select
   b ; stop
[]
   c ; stop
end select
```



Sémantique de « select »

$$\frac{i \in 1..n \qquad \{ B_i \}\ \rho\ -l\rightarrow_L \{ B_i' \}\ \rho'}{\{ \textbf{select}\ B_1\ []\ \ldots\ []\ B_n\ \textbf{end select} \}\ \rho\ -l\rightarrow_L \{ B_i' \}\ \rho'}$$

Rem :

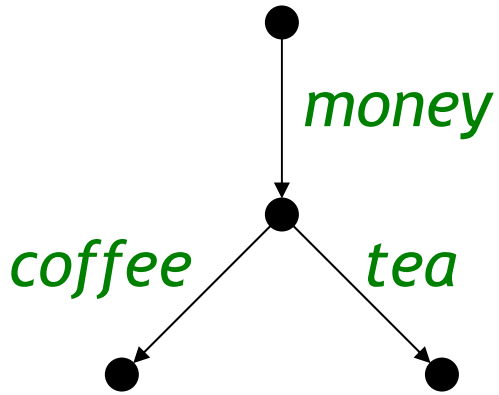
- n'existe pas dans la partie données
- après le choix, tous les autres comportements disparaîssent (on s'est engagé dans une branche du choix)

Choix externe / interne

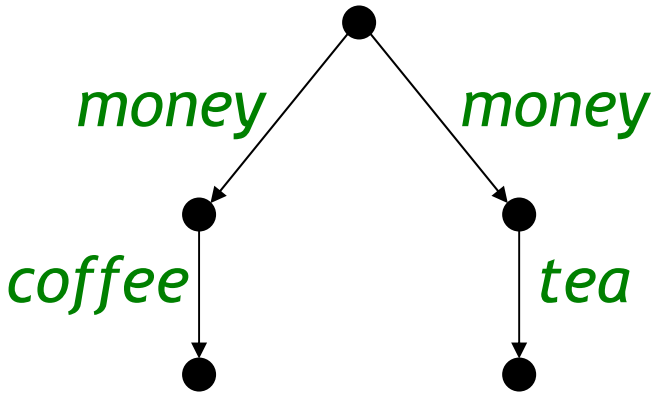
2 sémantiques de choix dans la littérature

- Choix externe : dans **select** G_1; B_1 [] G_2; B_2 **end select**, l'environnement décide de la branche choisie (s'il accepte G_1 et G_2, alors choix non-déterministe)
- Choix interne : le programme décide

- Ex. : distributeur de boissons



choix externe (user)



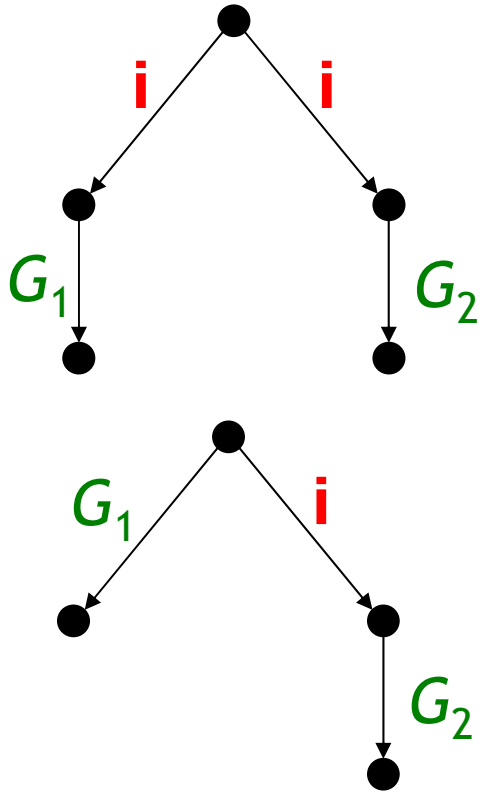
choix interne (machine)

Action interne (« i »)

- En LNT, la porte spéciale **i** (syntaxe concrète pour τ) désigne un événement interne sur lequel l'environnement n'a pas d'influence :

```
select
    i ; G1 ; stop
[]
    i ; G2 ; stop
end select

select
    G1 ; stop
[]
    i ; G2 ; stop
end select
```



choix interne, non visible par l'environnement : le programme peut toujours choisir **i**

Variables

- LNT permet de définir des variables pour mémoriser les résultats des expressions
- L'opérateur « **var** » (déclaration de variable) :

  var X_1:T_1, …, X_n:T_n **in** *B* **end var**

  déclare les variables X_1, …, X_n, qui sont visibles dans *B*
- Les variables prennent leur valeur par filtrage (**case**, communication) et par affectation (**:=**)

Sémantique de « := »

- **Affectation déterministe**

$$\frac{\{V\}\ \rho \rightarrow_e v}{\{X := V\}\ \rho \text{ –exit}\rightarrow_b \{\textbf{stop}\}\ \rho + [X := v]}$$

Rem : l'affectation de tableau est omise par simplicité

- **Affectation non-déterministe**

$$\frac{v \in T \qquad \rho' = \rho + [X := v] \qquad \{V\}\ \rho' \rightarrow_e \text{true}}{\{X := \textbf{any } T \textbf{ where } V\}\ \rho \text{ –exit}\rightarrow_b \{\textbf{stop}\}\ \rho'}$$

Rem : l'affectation non-déterministe est interdite dans la partie données, qui est totalement déterministe

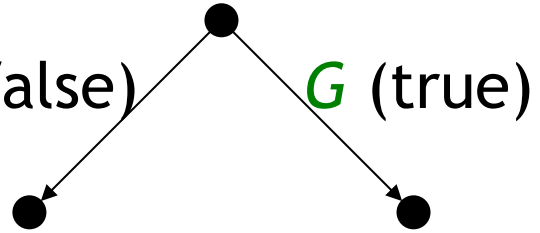
Remarques

Malgré ses opérateurs d'affectation, **LNT est similaire aux langages fonctionnels** :

- pas de variable non initialisée (*cf*. sémantique statique)
- pas de variables « globales » ou « partagées » entre fonctions ou processus
- chaque processus a ses propres variables locales
- communication uniquement par rendez-vous
- pas d'effets de bord

Exemples d'affectation non-déterministe

- *X* := **any** bool; *G* (*X*); **stop**

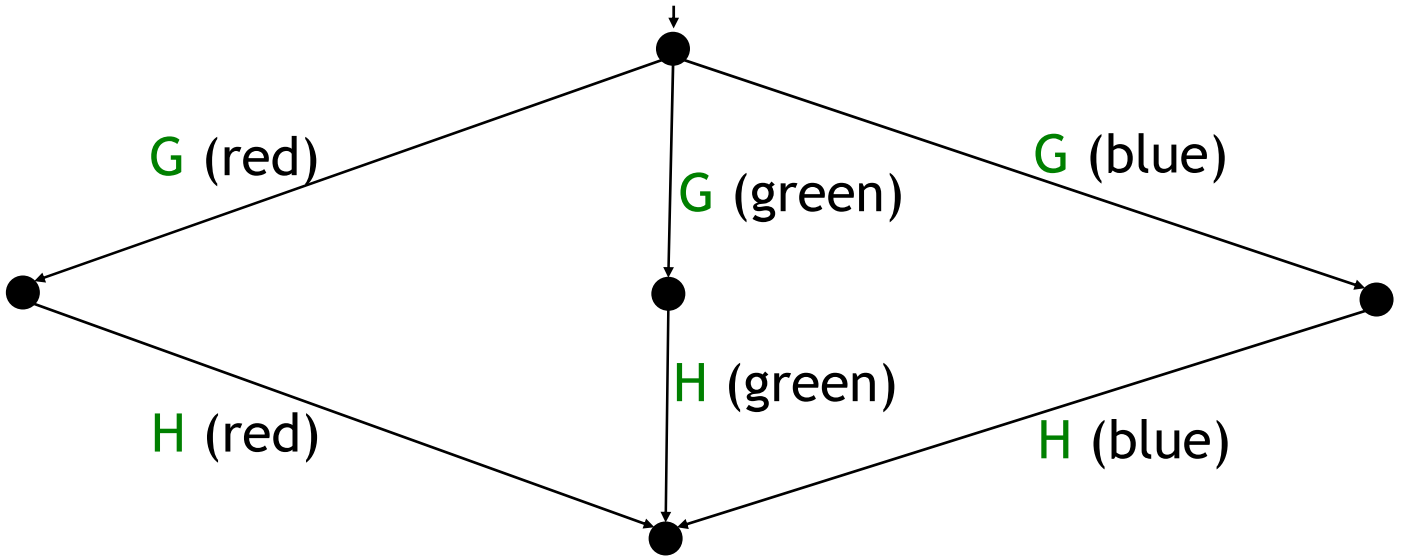


- Réception de valeur = cas particulier de **any**
  G (?*X*) ≡ *X* := **any** *T*; *G* (*X*) (avec *X* : *T*)
- Pour un type *T* fini (valeurs v_1, ..., v_n) :
  X := **any** *T* ≡ **select** *X* := v_1 [] ... [] *X* := v_n **end select**
- Génération d'une valeur aléatoire :

```
rand := any Nat;
if rand <= 10 then SEND (rand) end if
```

Exercice

- Donnez trois manières de décrire le comportement suivant, où red, green et blue sont les 3 valeurs d'un type Color :



Appel de processus

process Π [G_1, ..., G_n] (X_1:T_1, ..., X_m:T_m) **is**

B

end process

$$\frac{\{V_i\}\ \rho \rightarrow_e v_i \qquad \rho' = \rho + [X_1 := v_1, \ldots, X_m := v_m] \qquad \{\ B\ [G'_1/G_1, \ldots, G'_n/G_n]\ \}\ \rho' \text{ --}l\rightarrow_b \{\ B'\ \}\ \rho''}{\{\ \Pi\ [G'_1, \ldots, G'_n]\ (V_1, \ldots, V_m)\ \}\ \text{--}l\rightarrow_b \{\ B'\ \}\ \rho''}$$

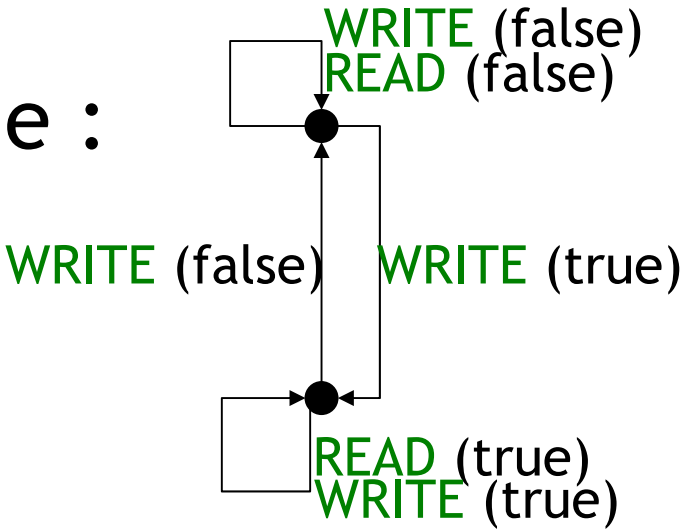
Remarques

- Cette sémantique montre qu'un processus de la forme **process** P [G : *Γ*] **is** P [G] **end process** a un comportement équivalent à **stop**

  On appelle cette situation « récursion non gardée » (appel récursif non précédé par une action)
- Seule la récursion terminale est autorisée
- Possibilité de passage de paramètre par référence (paramètres **out** et **inout**) ; voir la doc
- L'appel de processus est interdit dans la partie données

Exemple

Variable partagée booléenne :




```
process VARIABLE [READ, WRITE : bool] (b : bool) is
        select READ (b) [] WRITE (?b) end select;
        VARIABLE [READ, WRITE] (b)
end process
```

Sémantique de « while »

$$\frac{\{V\}\ \rho \rightarrow_e \text{true} \qquad \{\ B;\ \textbf{while}\ V\ \textbf{loop}\ B\ \textbf{end loop}\ \}\ \rho\ -l\rightarrow_b \{\ B'\ \}\ \rho'}{\{\textbf{while}\ V\ \textbf{loop}\ B\ \textbf{end loop}\}\ \rho\ -l\rightarrow_b \{\ B'\ \}\ \rho'}$$

$$\frac{\{\ V\ \}\ \rho \rightarrow_e \text{false}}{\{\ \textbf{while}\ V\ \textbf{loop}\ B\ \textbf{end loop}\ \}\ \rho\ -\textbf{exit}\rightarrow_b \{\ \textbf{stop}\ \}\ \rho}$$

Rem : LNT propose d'autres formes de boucles

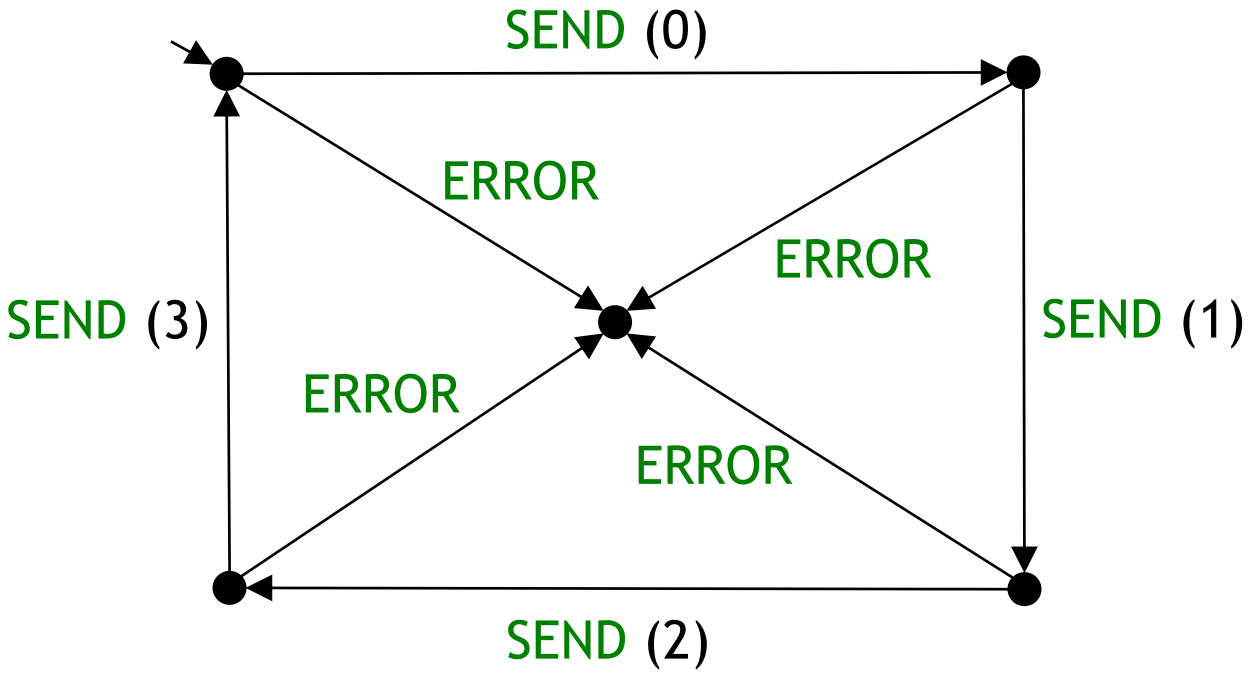
- **loop** *B* **end loop** ≡ **while** true **loop** *B* **end loop**
- **loop** *L* **in** *B* **end loop - break** *L* : voir le manuel pour la sémantique détaillée

Exemple

```
process VARIABLE [READ, WRITE : bool]
                          (b_init : bool) is
   var b : bool in b := b_init;
      loop
            select READ (b) [] WRITE (?b) end select
      end loop
   end var
end process
```

Exercice

- Ecrire un processus LNT qui a le comportement suivant :



Sémantique de « par » (1/2)

N'existe pas dans la partie données

Défini par 3 règles SOS

1. Entrelacement

$$i \in 0..m \quad \{ B_i \}\ \rho \xrightarrow{-G\ (v_1, \ldots, v_n)\rightarrow_b}\ \{ B_i' \}\ \rho' \qquad G \notin \underline{G}$$

$$\{ \textbf{par } \underline{G} \textbf{ in } B_0 \ ||\ \ldots\ B_i\ \ldots\ ||\ B_m \textbf{ end par} \}\ \rho$$

$$\xrightarrow{-G\ (v_1, \ldots, v_n)\rightarrow_b}$$

$$\{ \textbf{par } \underline{G} \textbf{ in } B_0 \ ||\ \ldots\ B_i'\ \ldots\ ||\ B_m \textbf{ end par} \}\ \rho'$$

$\underline{G}$ abréviation de $G_1, \ldots, G_n$

Sémantique de « par » (2/2)

2. Synchronisation

$$\frac{\{ B_i \}\ \rho \xrightarrow{-G\ (v_1, \ldots, v_n)\rightarrow_b}\ \{ B_i' \}\ \rho_i \qquad G \in \underline{G} \qquad \rho' = \rho + (\rho_0 - \rho) + \ldots + (\rho_m - \rho)}{\{\ \textbf{par}\ \underline{G}\ \textbf{in}\ B_0 \ ||\ \ldots\ ||\ B_m\ \textbf{end par}\ \}\ \rho \xrightarrow{-G\ (v_1, \ldots, v_n)\rightarrow_b}\ \{\ \textbf{par}\ \underline{G}\ \textbf{in}\ B_0' \ ||\ \ldots\ ||\ B_m'\ \textbf{end par}\ \}\ \rho'}$$

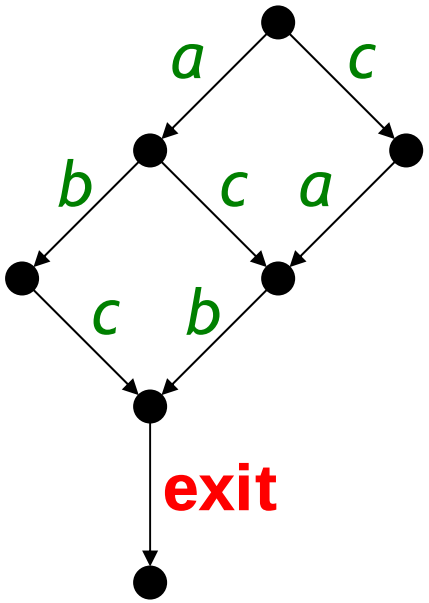
3. Terminaison synchronisée

$$\frac{\{ B_i \}\ \rho \xrightarrow{-\textbf{exit}\rightarrow_b}\ \{ B_i' \}\ \rho_i \qquad \rho' = \rho + (\rho_0 - \rho) + \ldots + (\rho_m - \rho)}{\{\ \textbf{par}\ \underline{G}\ \textbf{in}\ B_0 \ ||\ \ldots\ ||\ B_m\ \textbf{end par}\ \}\ \rho \xrightarrow{-\textbf{exit}\rightarrow_b}\ \{\ \textbf{stop}\ \}\ \rho'}$$

Terminaison synchronisée : exemple

Les comportements qui se terminent sont synchronisés sur la porte **exit** (ils se terminent en même temps)

par *a* ; *b* || *c* **end par**

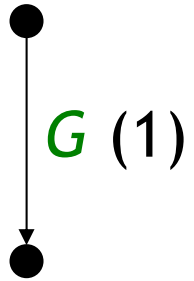


Exemples

Synchronisation par filtrage de valeurs (*value matching*) : deux comportements parallèles s'envoient les mêmes valeurs

par *G* **in** *G* (1) || *G* (1) **end par**

le RdV a lieu



par *G* **in** *G* (1) || *G* (2) **end par**

blocage (valeurs ≠)

par *G* **in** *G* (1 **of** nat) || *G* (1 **of** bit) **end par**

blocage (types ≠)

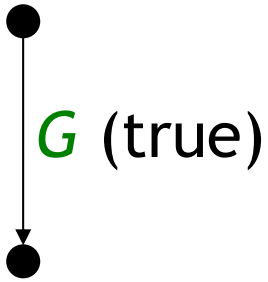
Exemples (suite)

Emission-réception :

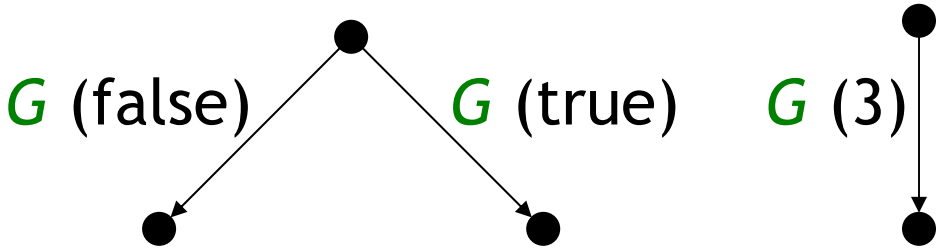
Soit *X* : bool

par *G* **in** *G* (?*X*) || *G* (true) **end par**

⇒ synchronisation



par *G* **in** *G* (?*X*) || *G* (3) **end par**



⇒ blocage : la sémantique de l'opérateur **par** exige que les deux étiquettes soient identiques (même nombre de paramètres, mêmes types et mêmes valeurs)

Exemples (suite)

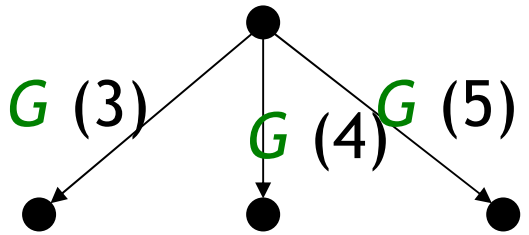
Synchronisation par production de valeurs (*value generation*) : deux comportements parallèles "reçoivent" des valeurs du même type

par *G* **in**

$G(?n_1)$ **where** $n_1 \leq 5$

|| $G(?n_2)$ **where** $n_2 > 2$

end par

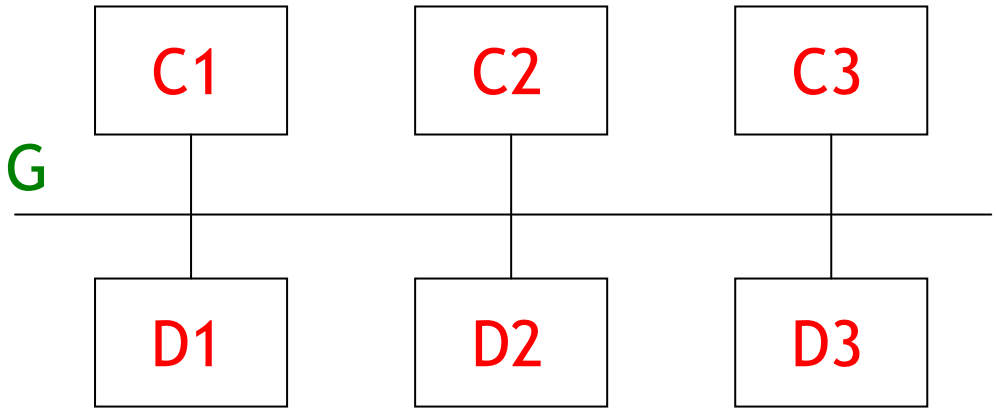


Multiplexage sur les portes

Ex. : bus avec périphériques

```
par G in
        par C1 [G] || C2 [G] || C3 [G] end par
||      par D1 [G] || D2 [G] || D3 [G] end par
end par
```

- chaque Ci et chaque Di proposent G (i)
  => RdV entre Ci et Di



Rem : permet d'éviter la création de portes en trop grand nombre.

Exercice

Dessiner le STE des processus suivants :

```
par G2 in
        G1; G2; G3
||
        G4; G2; G5
end par;
G6; stop
```

```
var X, Y, Z : bool in
par G in
        G (?X)
||
        G (?Y); Z := true
end par;
H (Z); stop
end var
```

Sémantique de « hide »

$$\frac{\{\ B\ \}\ \rho\ -G\ (v_1,\ \ldots,\ v_m)\rightarrow_b\ \{\ B'\ \}\ \rho' \qquad G \notin \underline{G}}{\{\ \textbf{hide}\ \underline{G}\ \textbf{in}\ B\ \textbf{end hide}\ \}\ \rho\ -G\ (v_1,\ \ldots,\ v_m)\rightarrow_b\ \{\ \textbf{hide}\ \underline{G}\ \textbf{in}\ B'\ \textbf{end hide}\ \}\ \rho'}$$

$$\frac{\{\ B\ \}\ \rho\ -G\ (v_1,\ \ldots,\ v_m)\rightarrow_b\ \{\ B'\ \}\ \rho' \qquad G \in \underline{G}}{\{\ \textbf{hide}\ \underline{G}\ \textbf{in}\ B\ \textbf{end hide}\ \}\ \rho\ -\textbf{i}\rightarrow_b\ \{\ \textbf{hide}\ \underline{G}\ \textbf{in}\ B'\ \textbf{end hide}\ \}\ \rho'}$$

$$\frac{\{\ B\ \}\ \rho\ -\textbf{exit}\rightarrow_b\ \{\ B'\ \}\ \rho'}{\{\ \textbf{hide}\ \underline{G}\ \textbf{in}\ B\ \textbf{end hide}\ \}\ \rho\ -\textbf{exit}\rightarrow_b\ \{\ B'\ \}\ \rho'}$$

Rem :

- aucun effet sur les portes hors de $\underline{G}$, qui restent visibles
- en LNT, l'action invisible s'appelle **i**
- n'existe pas dans la partie données

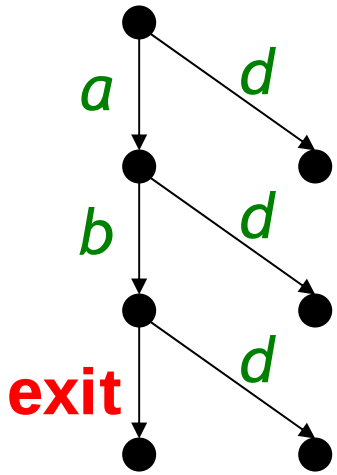
Opérateur « disrupt »

- LNT permet d'exprimer l'interruption d'un comportement par un autre (≈ exception) :

  disrupt B_1 **by** B_2 **end disrupt**

  signifie que l'exécution de B_1 peut être à tout instant interrompue et continuée avec B_2
- Exemple :

  disrupt *a* ; *b* **by** *d* ; **stop end disrupt**



- N'existe pas dans la partie données

Sémantique de « disrupt »

$$\frac{\{\ B_1\ \}\ \rho\ -G\ (v_1,\ \ldots,\ v_n)\rightarrow_b\ \{\ B_1'\ \}\ \rho'}{\{\ \textbf{disrupt}\ B_1\ \textbf{by}\ B_2\ \textbf{end disrupt}\ \}\ \rho\ -G\ (v_1,\ \ldots,\ v_n)\rightarrow_b\ \{\ \textbf{disrupt}\ B_1'\ \textbf{by}\ B_2\ \textbf{end disrupt}\ \}\ \rho'}$$

$$\frac{\{\ B_1\ \}\ \rho\ -\textbf{exit}\rightarrow_b\ \{\ B_1'\ \}\ \rho'}{\{\ \textbf{disrupt}\ B_1\ \textbf{by}\ B_2\ \textbf{end disrupt}\ \}\ \rho\ -\textbf{exit}\rightarrow_b\ \{\ B_1'\ \}\ \rho'}$$

$$\frac{\{\ B_2\ \}\ \rho\ -l\rightarrow_b\ \{\ B_2'\ \}\ \rho'}{\{\ \textbf{disrupt}\ B_1\ \textbf{by}\ B_2\ \textbf{end disrupt}\ \}\ \rho\ -l\rightarrow_b\ \{\ B_2'\ \}\ \rho'}$$

Module et point d'entrée LNT

- Un programme LNT est défini dans un module de même nom que le fichier qui le contient
- Possibilité d'importer des modules
- Un processus nommé MAIN (sans paramètres valeur) définit le point d'entrée du programme
- **Exemple** :

```
module mon_module (module_1, module_2) is
    ...
    process MAIN [...] is ... end process
end module
```

modules importés

définitions des types, fonctions, processus

TP-TD de LNT

- Séance du 2 déc. (W. Serwe) en salle machine
- A préparer d'ici là :
  - fichiers à récupérer sur ensisun dans /home/perms/serwe/TD-Cadp
  - sourcer /home/perms/serwe/TD-Cadp/Cadp_env.sh pour utiliser CADP
  - mail à {Frederic.Lang,Wendelin.Serwe}@inria.fr en cas de problème
  - préparer les questions du fichier sujet.pdf

Exemple : algorithme de Peterson (exclusion mutuelle)

Pseudocode

var d0 : bool := false { lue par P1, écrite par P0 }
var d1 : bool := false { lue par P0, écrite par P1 }
var t ∈ {0, 1} := 0 { lue/écrite par P0 et P1 }

```
loop forever { P0 }
1 : { snc0 }
2 : d0 := true
3 : t := 0
4 : wait (d1 = false or t = 1)
5 : { sc0 }
6 : d0 := false
end loop
```

```
loop forever { P1 }
1 : { snc1 }
2 : d1 := true
3 : t := 1
4 : wait (d0 = false or t = 0)
5 : { sc1 }
6 : d1 := false
end loop
```

Modélisation des variables d0, d1

- chaque variable : instance d'un même processus D
- lecture et écriture : RdV sur les portes R et W

```
process D [R, W : bool] is
  var b : bool in
    b := false;
    loop select R (b) [] W (?b) end select end loop
  end var
end process
```

channel bool is (bool) end channel

- d0 ≡ D [R0, W0], d1 ≡ D [R1, W1]

Modélisation de la variable t

- variable t : instance d'un processus T
- lecture et écriture : RdV sur les portes R et W

```
process T [R, W : nat] is
  var n : nat in
    n := 0;
    loop select R (n) [] W (?n) end select end loop
  end var
end process
```

channel nat is (nat) end channel

- t ≡ T [RT, WT]

Modélisation des processus P0 et P1

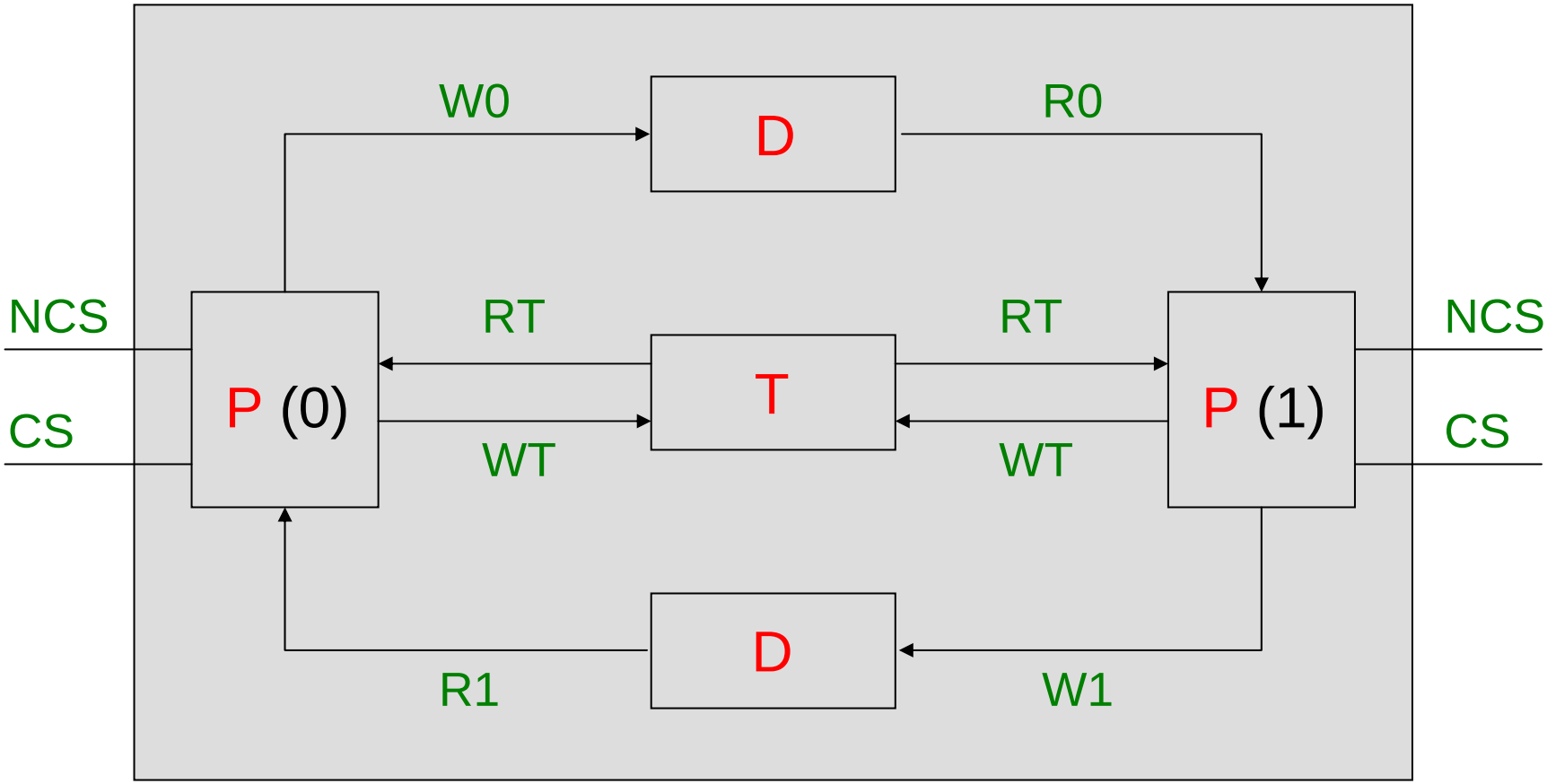
- processus : instances d'un même processus P
- index du processus : paramètre de P
  - P0 ≡ P [R0, W0, R1, W1, RT, WT, NCS, CS] (0)
  - P1 ≡ P [R1, W1, R0, W0, RT, WT, NCS, CS] (1)

```
channel none is ()
end channel
```

```
process P [Rm, Wm, Rn, Wn : bool, RT, WT : nat, NCS : none, CS : nat]
   (m : nat) is

   var dn : bool, t : nat in
      loop
         NCS;
         Wm (true);
         WT (m);
         loop wait in
            Rn (?dn);
            RT (?t);
                                  .../...
                                          if not (dn) or (t != m) then
                                             CS (m);
                                             Wn (false);
                                             break wait
                                          end if
                                       end loop
                                    end loop
                                 end var
                              end process
```

Architecture du système



Architecture du système (suite)

```
hide R0, W0, R1, W1 : bool, RT, WT : nat in
  par R0, W0, R1, W1, RT, WT in
    par
      P [R0, W0, R1, W1, RT, WT, NCS, CS] (0)
    || P [R1, W1, R0, W0, RT, WT, NCS, CS] (1)
    end par
  ||
    par
      T [RT, WT]
    || D [R0, W0]
    || D [R1, W1]
    end par
  end par
end hide
```

Remarques

- LNT est plus concis que les ACs: paramétrisation des processus, communication de valeurs
- En général, il existe plusieurs façons d'exprimer la mise en parallèle des composants du système
- Pour les comportements cycliques : choix entre style itératif (boucle **loop**) et style récursif (appel de processus)

Spécification descendante
(par raffinements successifs)

- identifier les processus LNT qui s'exécutent en parallèle (process MAIN)
- identifier les portes visibles et les portes cachées
- identifier les synchronisations entre processus ; attention aux RdV à 2 sur une porte G où $n > 2$ processus sont connectés (discriminer avec une offre d'émission *V*)
- identifier les valeurs initiales des paramètres de chaque processus
- décrire le comportement (habituellement séquentiel) de chaque processus

Algèbres de processus temporisées

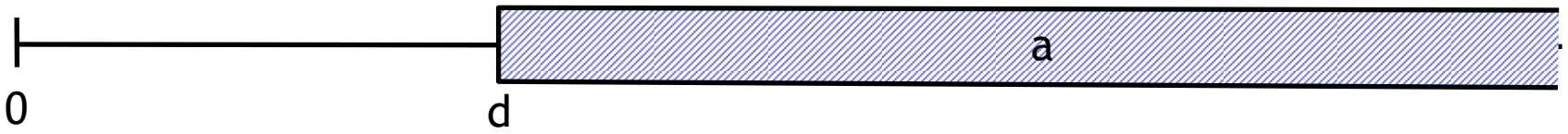
- AdP + contraintes de temps
- Extensions des AdP non-temporisées
  - Timed CCS, Timed CSP, extensions de ACP, etc.
  - Plusieurs extensions de LOTOS : T-LOTOS, ET-LOTOS, RT-LOTOS, E-LOTOS, etc.
- Extension temporisée de LOTOS NT

Extension temporisée de LOTOS NT

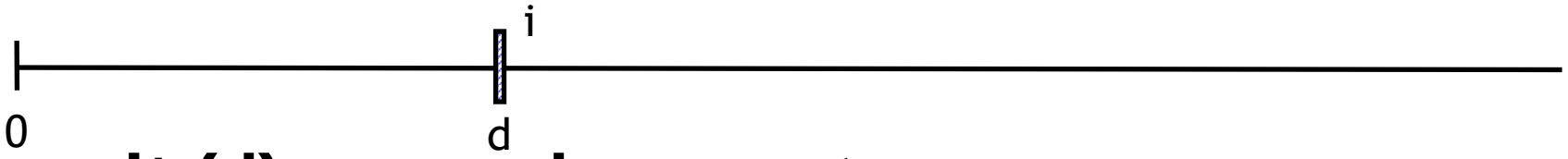
- Non supportée par des outils pour le moment
- Extensions :
  - « G (O_1, …, O_n) [@X] **where** E » :
    - X = temps écoulé depuis que la communication est possible
    - E peut inclure des conditions sur la valeur de X
  - « **wait** (E) » : délai déterministe
  - Toute action interne est urgente

Opérateurs temporisés : exemples

- **wait** (d); a

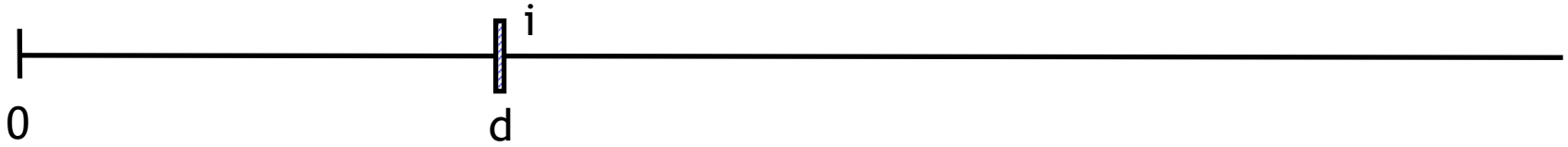


- **hide** a **in wait** (d); a



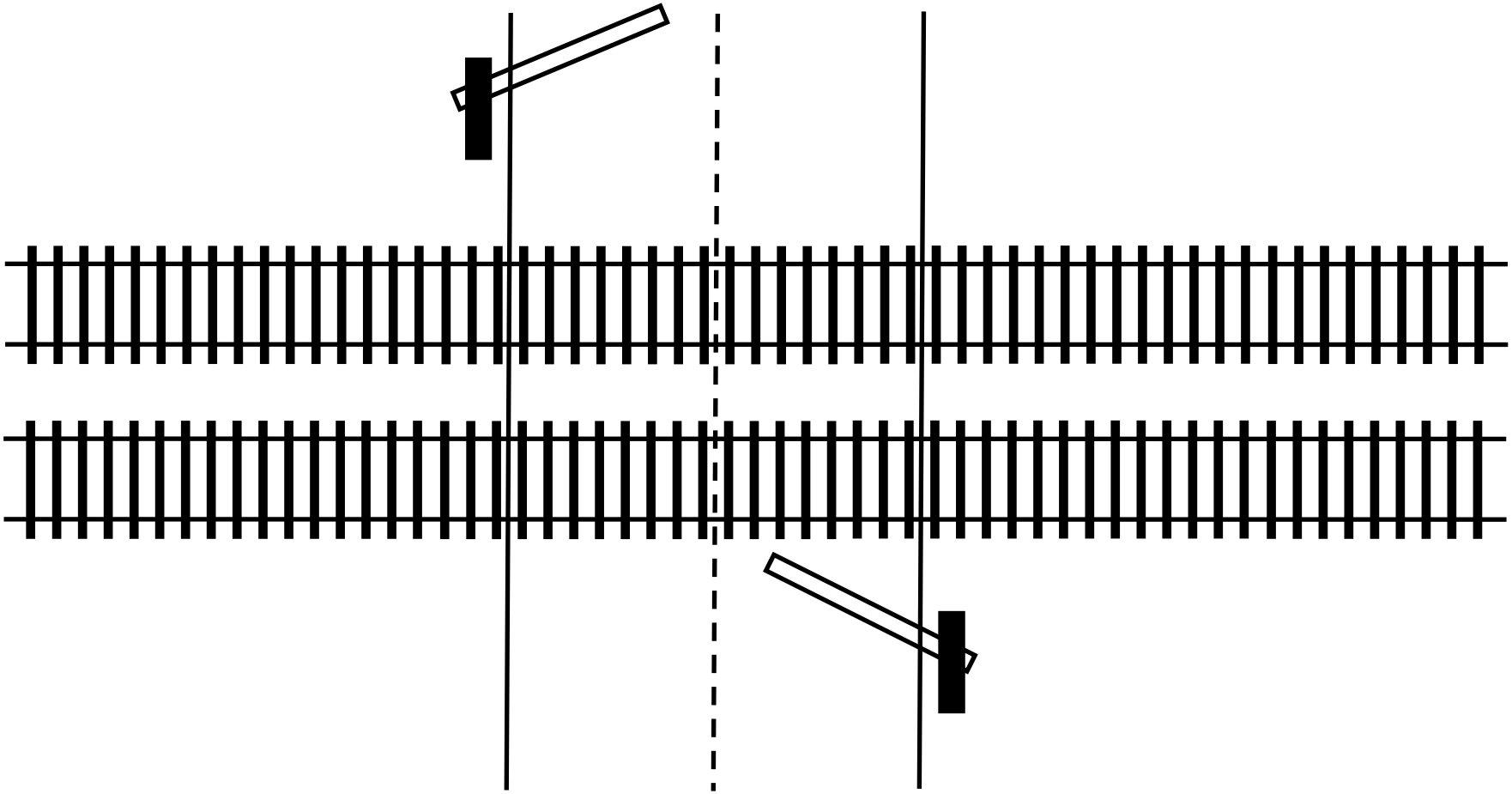
- **wait (d);** a @x **where** x ≤ t

- **hide** a **in wait (d);** a @x **where** x ≤ t



a ou i est possible

Exemple : passage à niveau (1/2)



Exemple : Passage à niveau (2/2)

- Composants : train, gate, controller
- Événements :
  approach, into, leave, lower, raise, down, up
- Contraintes :
  - train atteint le passage au moins 300 s après approach
  - train quitte le passage au plus 500 s après approach
  - controller envoie lower 100 s après approach
  - controller envoie raise au plus 100 s après leave
  - gate répond à lower par down en moins de 100 s
  - gate répond à raise par up après 100 à 200 s

Passage à niveau (1/4)

```
process MAIN [...] is
   par
            approach, leave →
                  Train [approach, into, leave]
   ||       approach, leave, lower, raise →
                  Controller [approach, lower, leave, raise]
   ||       lower, raise →
                  Gate [lower, raise, down, up]
   end par
end process
```

Passage à niveau (2/4)

```
process Controller [approach, lower, leave, raise : none] is
    var x : time in
        loop
                approach;
                wait (100);
                lower;
                leave;
                raise @x where x ≤ 100
        end loop
    end var
end process
```

- controller envoie lower 100 s après approach
- controller envoie raise au plus 100 s après leave

Passage à niveau (3/4)

```
process Gate [lower, raise, down, up : none]  is
    var x : time in
        loop
                lower;
                down @x where x ≤ 100;
                raise;
                up @x where (x ≥ 100) and (x ≤ 200)
        end loop
    end var
end process
```

- gate répond à lower par down en moins de 100 s
- gate répond à raise par up après 100 à 200 s

Passage à niveau (4/4)

```
process Train [approach, into, leave] : noexit :=
   var x, y : time in
      loop
            approach;
            into @x where x ≥ 300;
            leave @y where y+x ≤ 500
      end loop
   end var
endproc
```

- train atteint le passage au moins 300 s après approach
- train quitte le passage au plus 500 s après approach