Expression

- Term with a unique value in a given context
- Several kinds of expressions:
  - Literal constants: "Joe", 46, true, 'a', …
  - Variables (declared in the context): p, tab, x, …
  - Constructor application: tuple ("Joe", 46, male)
  - Function application: fact (3)
  - Array element access: tab[x]
  - Field access (if type defined **with** get): p.gender
  - Field update (if type defined **with** set):
    p.{name -> "Marilyn", gender -> female}
  - Type disambiguation: 46 **of** nat, 1 **of** int

Symbol overloading

- Functions or constructors may have same names but different types

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

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

- Same for the predefined functions

  function + (b1, b2 : **bool**) : **bool is** …

- **of** can be used to disambiguate:
  odd (1 **of nat**) vs. odd (1 **of int**)

Patterns

- Term that represents a set of possible values
- Allows pattern-matching inherited from functional languages (ML, Haskell, ...)
- Several kinds of patterns:
  - Literal constant: ″Joe″, 46, true, ‘a’, …
  - Variable (declared in the context): p, tab, x, …
  - Application of a constructor to patterns: tuple (″Joe″, 46, male)
  - Wildcard: **any** pers, **any nat**
  - Conditional pattern: tuple (**any string**, x, male) **where** x > 40

Pattern matching

- **case** instruction

```
case p var name : string, x : nat in
   tuple (name, x, any gender) where x < 18 ->
            return name
| tuple (any string, any nat, female) ->
            return " Missis "
| tuple (any string, any nat, any gender) ->
             return " Mister "
end case
```

- Rule priority in the reading order
- Variables occurring in the pattern are assigned by pattern matching: if p = tuple (« Toto », 7, male) then first rule applies and name takes the value « Toto »

Control part

- Allows processes to be defined
- Super-set of the data part:
  - All statements of the data part are available in the control part
  - Except **return** (reserved to the data part)
- Statements are added, describing:
  - nondeterminism
  - asynchronous parallelism
  - communication
  - action hiding

forbidden in the data part!

- Semantics: process $\rightarrow$ LTS (cf. appendix + [doc])

Event and channel

- **Event**: communication point between a process and its environnement (synchronisation, data exchange)
- **Channel**: constraint on the type of values that can be exchanged on an event
- Predefined channels:
  - **any**: no type constraint
  - **none**: no value can be exchanged
- User-defined channels: list of tuples

  Example : **channel** T **is** (**nat**), (**bool**, **bool**), () **end channel**
  either natural number or two booleans or no value

Process definition

Analogous to a function definition

- Similarities: value parameters **in**, **out**, **in out**
- Differences:
  - event parameters (typed by a channel)
  - No return value/type (no **return**)
  - No symbol overloading
- Example:

process SEMAPHORE [REL : **none**, REQ : **none**]
(locked : **bool**) **is**
... (* statements describing the process behaviour *)
end process

Remarks

- An LNT process is a « black box » with outside communication points

- The control part allows these black box behaviours and their parallel composition to be described

Statements existing both in the data part and in the control part (1/2)

- **Declaration of local variables:**
  var x, y **: nat,** tab : parray **in** … **end var**
- **Null operation** (passes to the next statement):
  null
- **Sequential composition:** ;
  statement separator rather than statement ending
- **Deterministic variable assignment:** x := 1
- **Array assignment:** tab[x] := p
- **Case pattern matching:** case … end case

Statements existing both in the data part and in the control part (1/2)

- If-then-elsif-else:
  if x > 10 **then** … **elsif** x > 1 **then** … **else** … **end if**
- Non-breakable loop: **loop** … **end loop**
- Breakable loop:
  loop L **in** … **break** L … **end loop**
- While loop:
  while x > 0 **loop** … x := x – 1 … **end loop**
- For loop:
  for x := 0 **while** x < 10 **by** x := x + 1 **loop**
  …
  end loop

Statements existing only in the control part

Originality with respect to classical algorithmic languages:

1. **Process call**
2. **Communication action (through an event)**
3. **Inaction (stop)**
4. **Nondeterministic assignment (any)**
5. **Nondeterministic choice (select)**
6. **Parallel composition (par)**
7. **Event hiding (hide)**

(1) Process call

- Actual value parameters: Similar to function calls
- No return value
- Actual event parameters: declared events
- No overloading
- **Example** :

  SEMAPHORE [R1, Q1] (false)

  where R1 and Q1 are declared events

(2) Communication action: definition

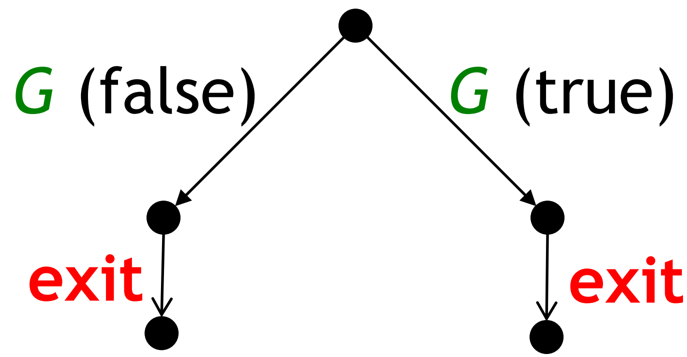
- Structured action
- Event possibly followed by one or several *communication offers*:
  - Value emission: **!***V* (or *V*), with V an expression
  - Value reception: **?***P*, with *P* a pattern
- Possibility to use a Boolean guard (**where**) to constrain the domain of exchanged values
- Possibility to freely mix emission and reception offers in the same communication action
- Example : SND (**?**X, 1 **of** nat) **where** X != me

(2) Communication action: semantics

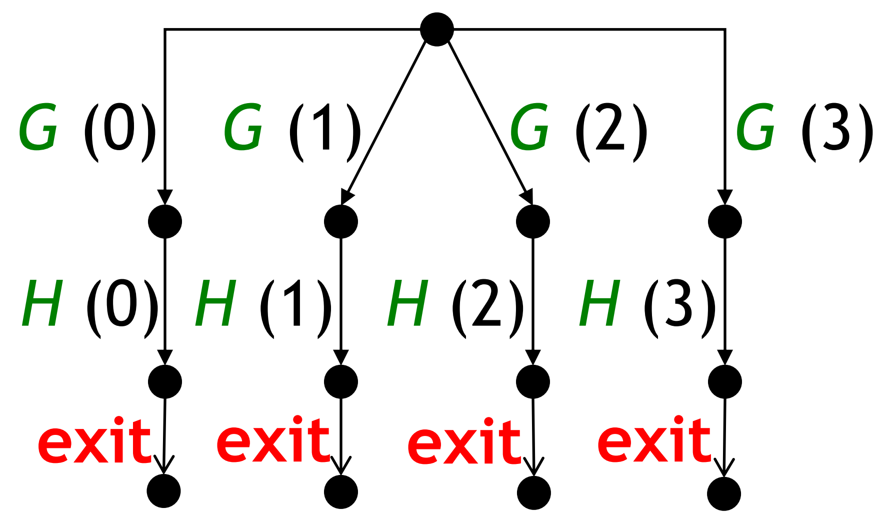
- LTS semantics: choice between actions when enumerating all values possibly exchanged
- Each action is followed by a special **exit** action, which indicates that the control can pass to the next instruction
- The **exit** action is consumed by sequential composition: a single final **exit** remains for every instruction sequence that terminates normally

(2) Communication action: examples

G (?*X*)
with *X*:bool



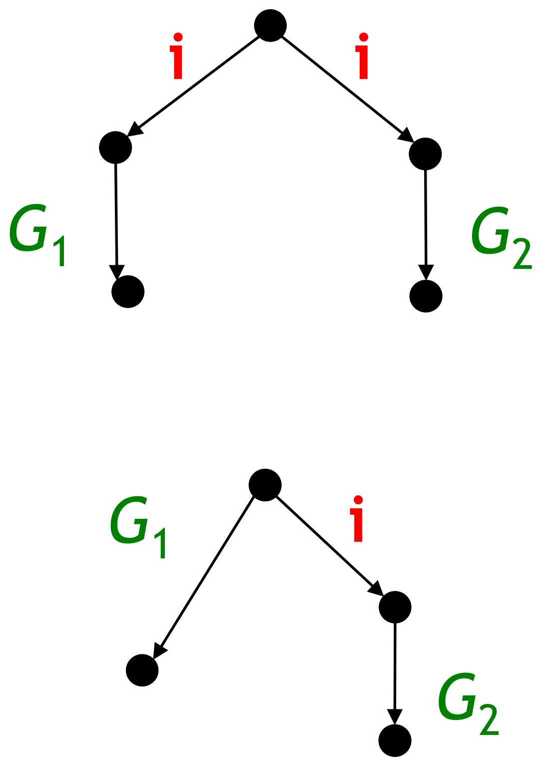
G (?*X*) **where** *X* < 4; H (*X*)
with *X*:nat



Remark: the semantics handles data reception by iterating over all values possibly received. Boolean guards allow the value domain of the reception variables to be constrained

(2) Communication action: the internal action « i »

- Equivalent of τ in CCS, noted **i** (predefined symbol)
- No communication offers
- Internal event that the environment cannot control



internal choice, not controllable by the environment: the program can always choose **i**

(3) Inaction

- **stop** statement, equivalent of **nil** in CCS
- Unlike **null**, control does not pass to the next statement: for each statement B

  stop; B = **stop**

- Example :

  if x > 0 **then** G **else stop end if**

  (abbrev. **only if** x > 0 **then** G **end if**)

  is different from

  if x > 0 **then** G **end if**

  which is equivalent to

  if x > 0 **then** G **else null end if**

(3) Inaction

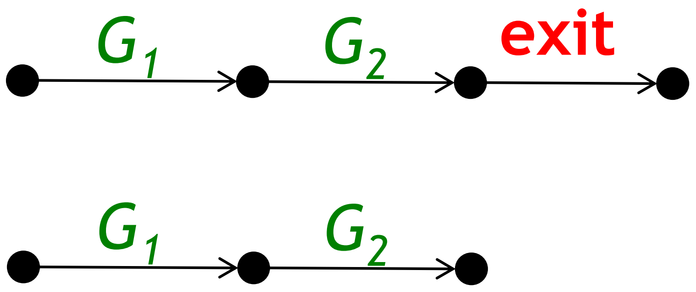
- If an instruction sequence terminates by **stop** then the **exit** action is consumed

  Example :

  G_1; G_2

  vs.

  G_1; G_2; **stop**



- Remark: “G **where** false” is equivalent to **stop**

(4) Nondeterministic assignment

- Statement allowing a set of values to be enumerated
- **Examples** :

  b := **any bool**

  x := **any nat where** x < 10
- Remark:

  G (?x) **where** x < 10

  is semantically equivalent to

  x := **any nat where** x < 10;

  G (x)

(5) Nondeterministic choice

- n-ary equivalent of the CCS binary operator +
- Choice between several execution branches
- Example :

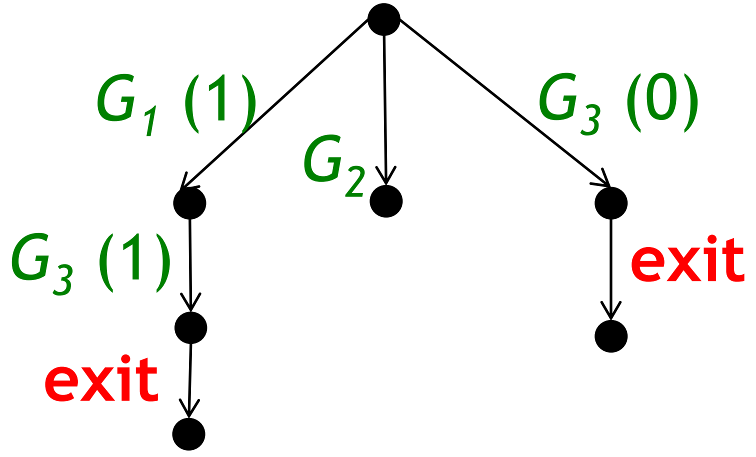
  x := 0;

  select

  x := 1; G_1 (x) **[]** G_2; ***stop*** **[] null**

  end select;

  G_3 (x)



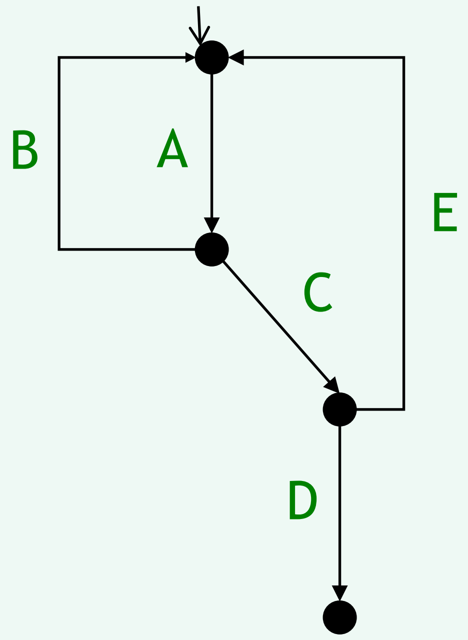
Exercise

- Complete the process P so that its behaviour is equivalent to the LTS depicted below, using LNT choice, sequence, and either loop or recursive process call

process P [A, B, C, D, E : **none**] **is**

…

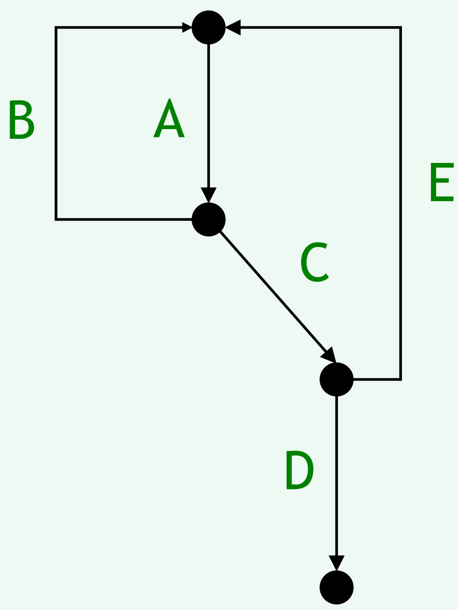
end process



Solutions

Imperative style (loop) :

```
process P [A, B, C, D, E : none] is
   loop
      A;
      select
         B
      []
         C;
         select
            D; stop
         []
            E
         end select
      end select
   end loop
end process
```



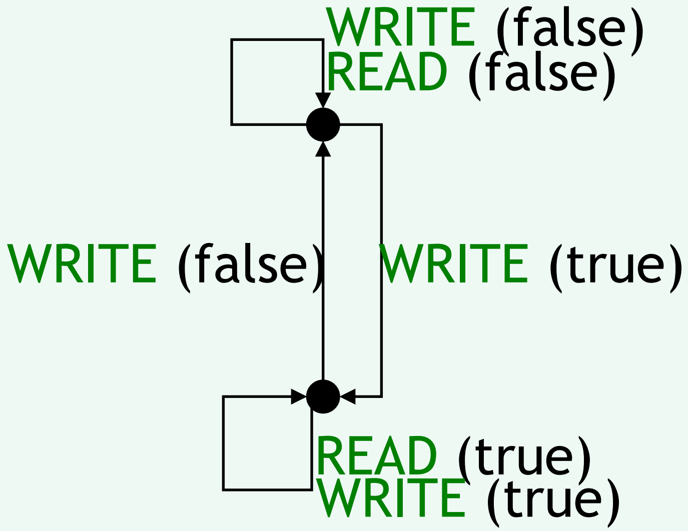
Recursive style:

```
process P [A, B, C, D, E : none] 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
```

Exercise: Boolean shared variable

Complete process VARIABLE below

```
process VARIABLE [READ, WRITE : bool] (b : bool) is
    ...
end process
```

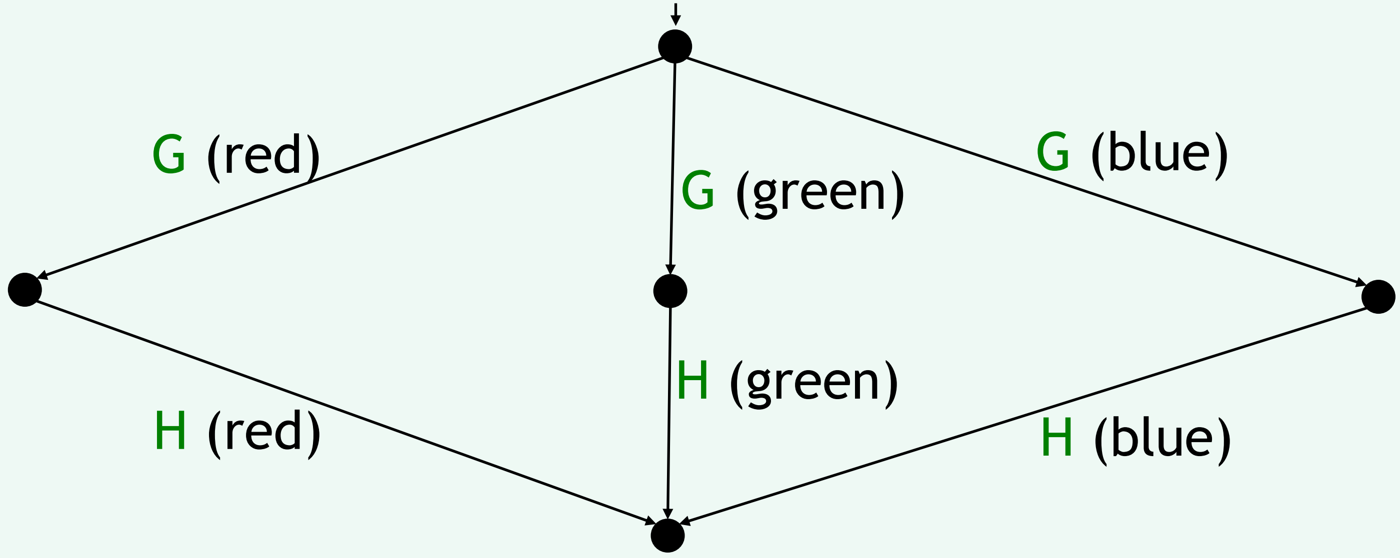


Solution

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

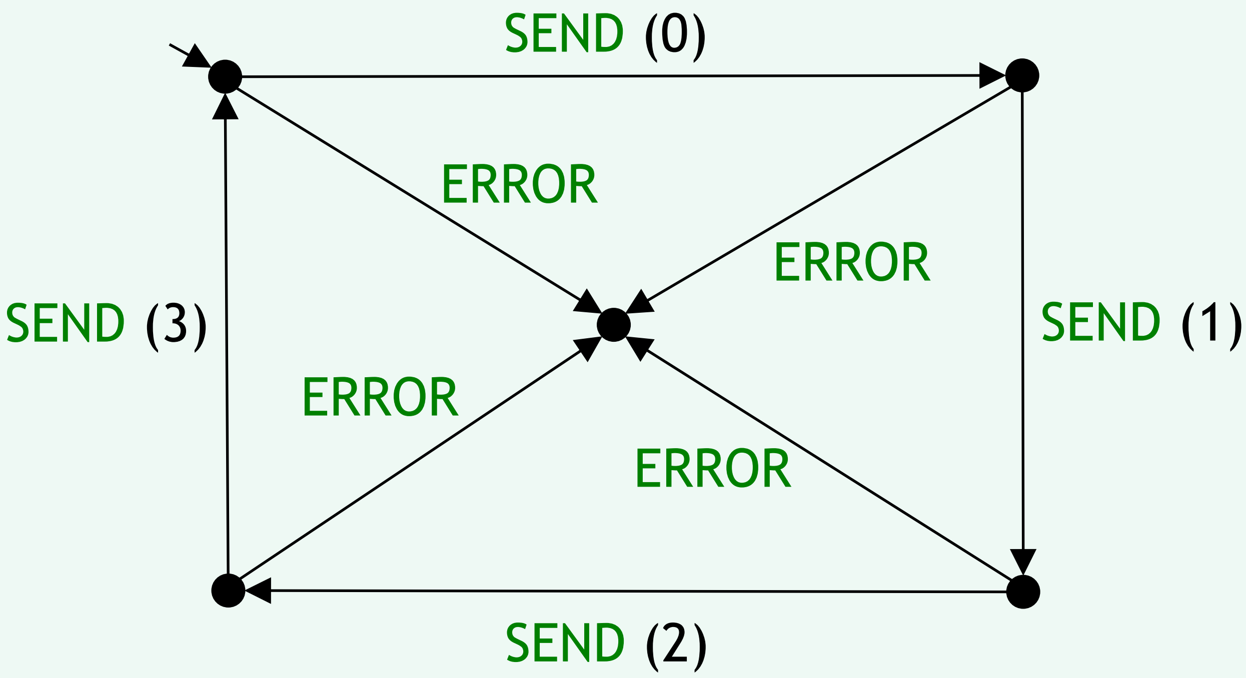
Exercise

- Give three ways to describe the following behaviour:



Exercise

- Write an LNT process, which has the following behaviour:



(6) Parallel composition

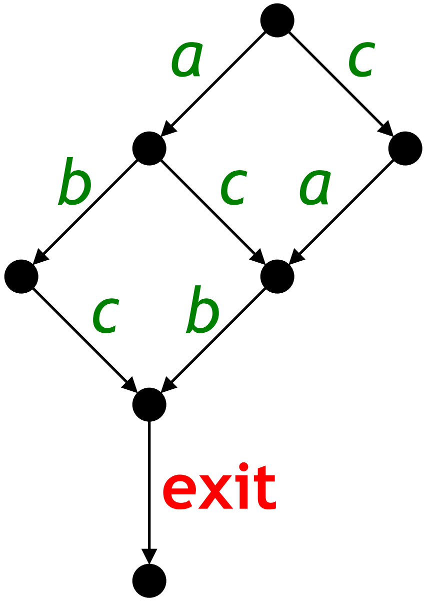
- n-ary generalisation of the $\otimes$ operator of CA
- Parameterised by the events to be synchronised
- Two (combinable) ways to express synchronisations
  1. Global synchronisation list
     Ex. : **par** G_1, G_2 **in** P_1 || P_2 || P_3 **end par**
     The three of P_1, P_2 and P_3 must synchronise on G_1 and G_2
     No synchronisation on other events
  2. Synchronisation interfaces
     Ex. : **par** G_1,G_2-> P_1 || G_2,G_3-> P_2 || G_3,G_1-> P_3 **end par**
     P_1 and P_3 must synchronise on G_1, P_1 and P_2 on G_2, and P_2 and P_3 on G_3

(6) Parallel composition: remark

If the parallel processes terminate, they must do it all together (*join*) by a synchronisation on **exit**.

Example :

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



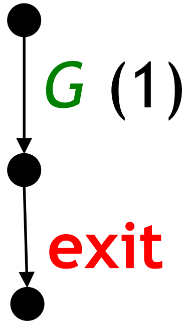
(6) Parallel composition: process communication (1/3)

The communication mode is value-matching synchronisation: two parallel processes agree on the values to be exchanged

Examples :

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

the synchronisation happens



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

deadlock (= **stop**) because values differ

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

deadlock because types differ

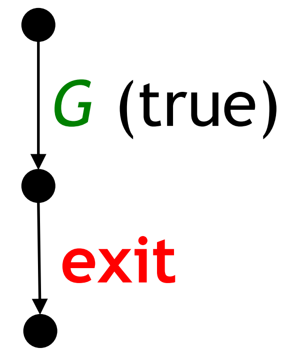
(6) Parallel composition: process communication (2/3)

Emission-reception:

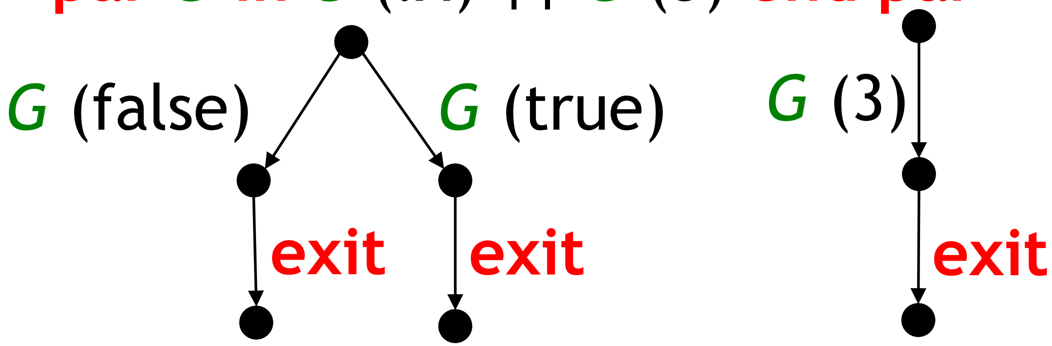
Let *X* : bool

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

⇒ synchronisation



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



⇒ deadlock: the semantics of **par** requires that both actions are the same (same number of parameters, same types, and same values)

(6) Parallel composition: process communication (3/3)

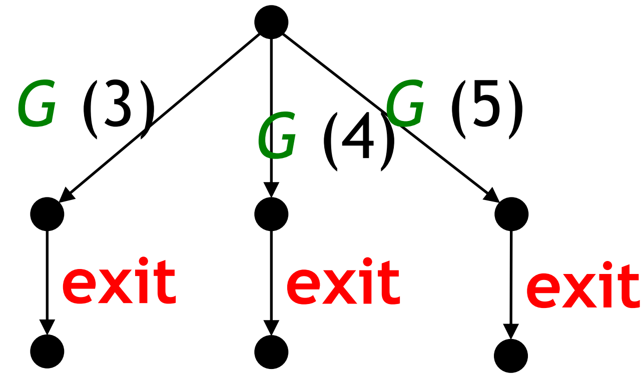
Synchronisation by value generation: two parallel behaviours "receive" values of the same type

par G **in**

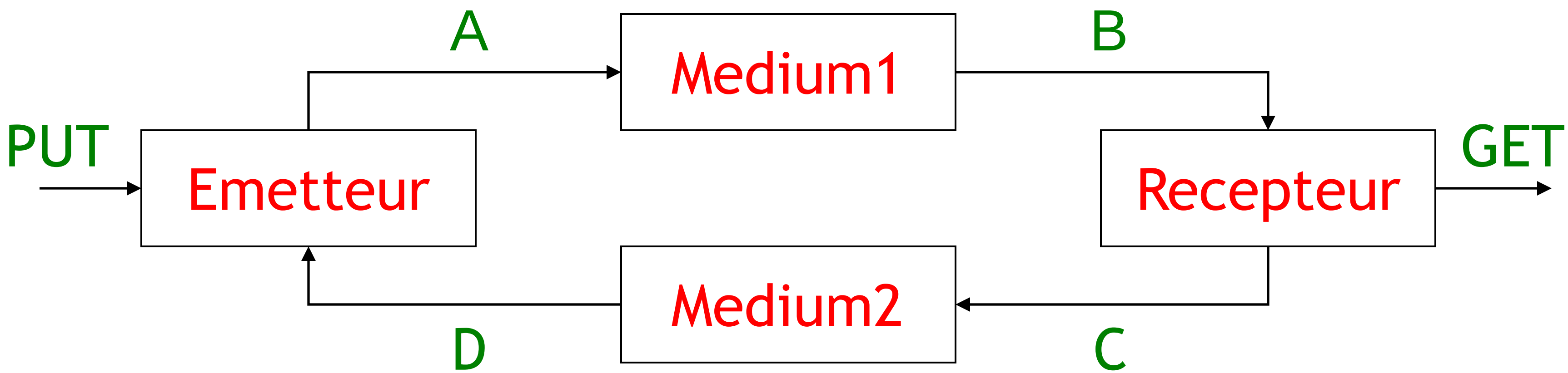
$G(?n_1)$ **where** $n_1 \leq 5$

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

end par



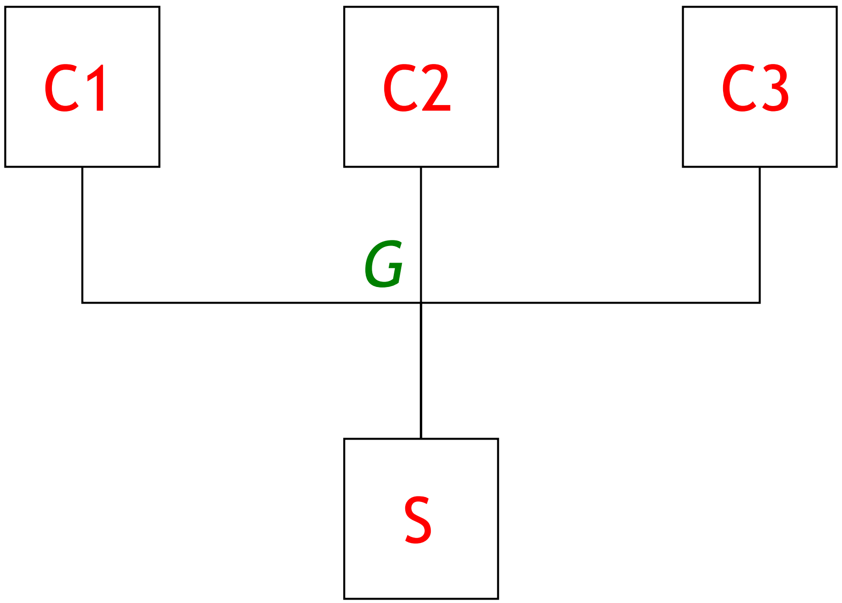
(6) Parallel composition: communication protocol example




```
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
```

(6) Parallel composition: n-ary synchronisation example

The LNT operators allow more than 2 processes to communicate on the same event (more powerful than CCS).



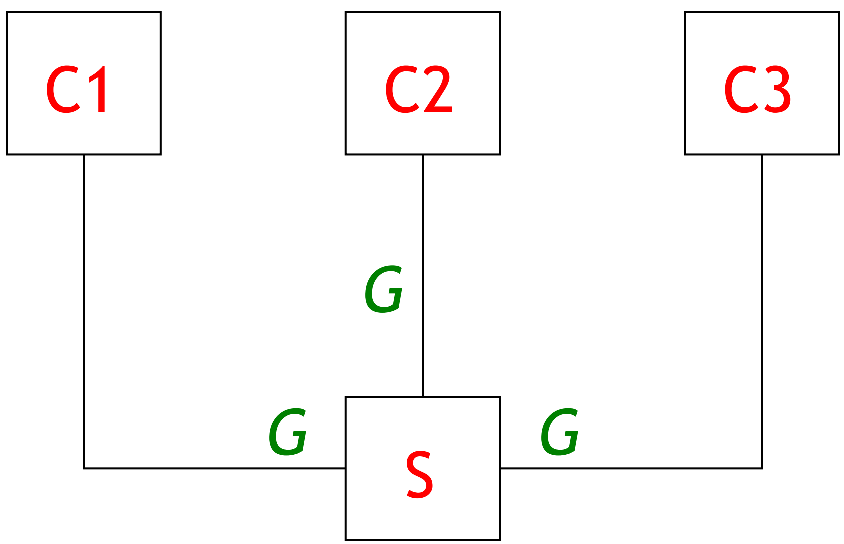
Example (client-server) :

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

The four (clients and server) processes synchronise all together on *G*.

(6) Parallel composition: concurrent binary synchronisations example

The **par** operator allows concurrent binary synchronizations on a same event to be modeled.



Example (client-server) :

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

Each of the client processes can access the server by a binary synchronisation on *G*. If several clients want to execute *G*, there is nondeterministic choice.

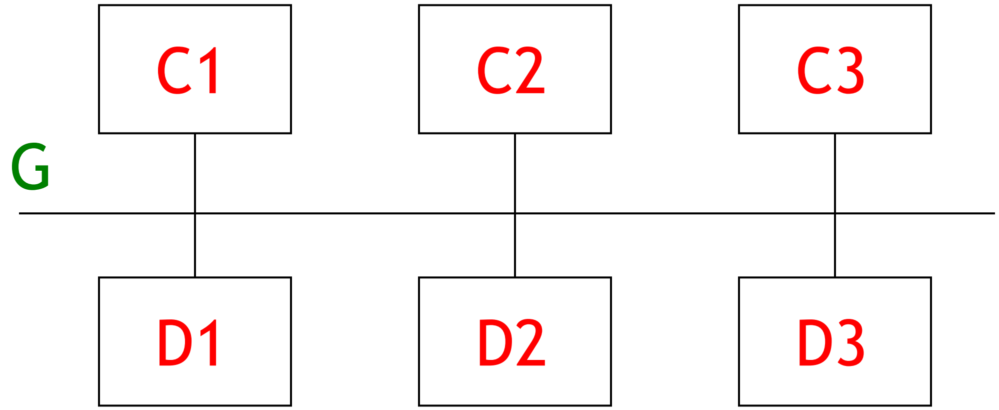
(6) Parallel composition: event multiplexing example

Bus with peripherics

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

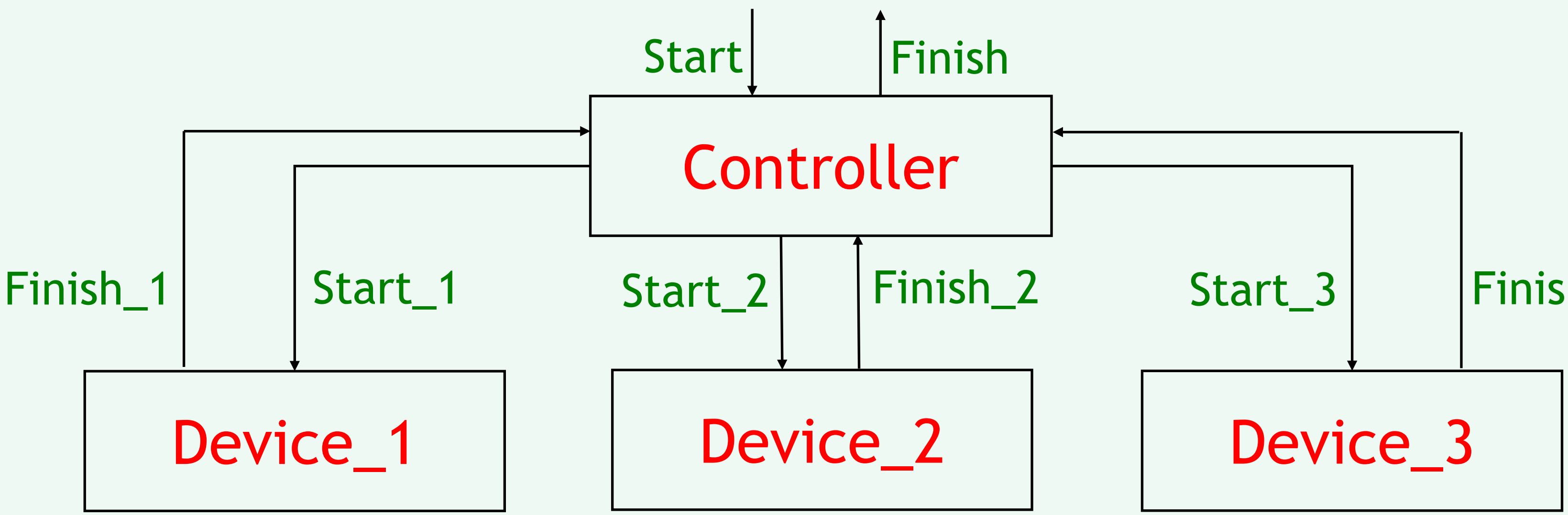
- Each Ci and each Di propose G (i)
  $\Rightarrow$ RdV between Ci and Di

Remark: allows the creation of numerous events to be avoided.



Exercise

- Write a parallel composition statement corresponding to the following parallel architecture:



Exercise

Draw the LTS of the following processes:

```
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
```

(7) Event hiding

- Unlike CCS, events/communication actions are not renamed into i after synchronisation
  $\Rightarrow$ other processes can participate

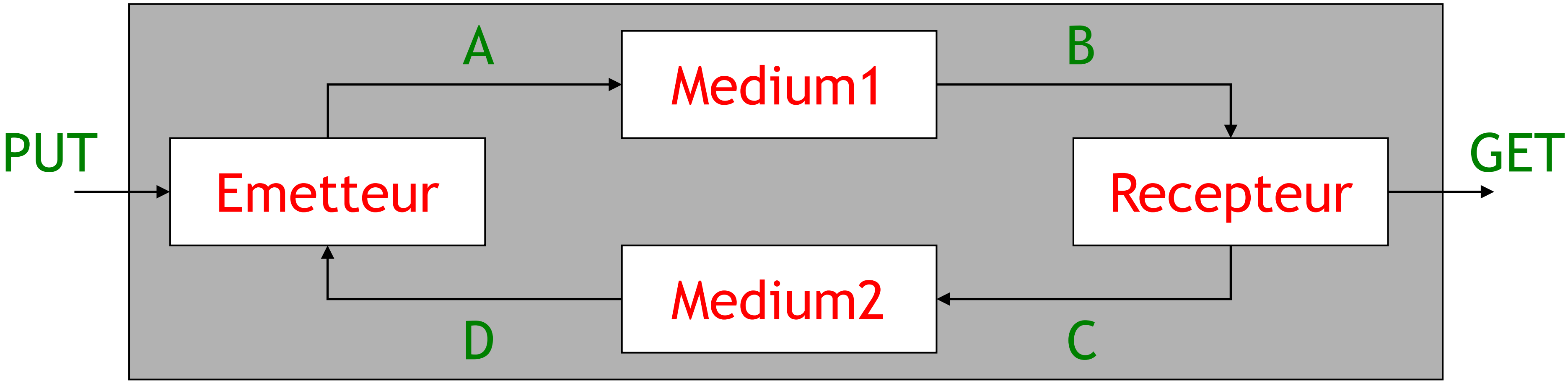
- To avoid this, events can be **hidden**, i.e., renamed into i, using the hide statement.

  Example :

  hide G_1, G_2 : none, G_3 : any in P end hide

  means that in P, all occurrences of (actions on events) G_1, G_2, and G_3 are renamed into i

(7) Event hiding: example




```
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
```

Remarks

Despite its assignment statements, **LNT is similar to functional languages**:

- no uninitialized variable can be read (static semantic constraint)
- no built-in « global » or « shared » variables between functions or processes
- each process has its own local variables
- communication only by event synchronisation
- no side-effects

Static semantics (1/2)

Guarantees the well definition of programs

- **Binding:** evary variable/event must be declared (parameter, **var** ... **end var**, **hide** … **end hide**)
- **Typing:** strict, by name
- **Initialisation:** every variable must have been defined before being used (*cf*. Java)
- A variable defined / modified in a branch must not be used in the parallel branches

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

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

Static semantics (2/2)

Restrictions concerning recursive processes

- **Terminal** recursion only

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

- No recursion through par (no replication)

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

Module and LNT entry point

- An LNT program is defined in a module whose name is the same as the file that contains it
- Possibility to import modules
- A process named MAIN (without value parameters) defines the program entry point
- **Example :**

imported modules

module mon_module (module_1, module_2) **is**

    ...

definitions of types, functions, processes

    process MAIN [...] **is** ... **end process**

end module

Example : the Peterson algorithm (mutual exclusion)

Pseudocode

var d0 : bool := false { read by P1, written by P0 }
var d1 : bool := false { read by P0, written by P1 }
var t $\in$ {0, 1} := 0 { read/written by P0 and P1 }

loop forever { P0 }	**loop forever** { P1 }
1 : { snc0 }	1 : { snc1 }
2 : d0 := true	2 : d1 := true
3 : t := 0	3 : t := 1
4 : **wait** (d1 = false **or** t = 1)	4 : **wait** (d0 = false **or** t = 0)
5 : { sc0 }	5 : { sc1 }
6 : d0 := false	6 : d1 := false
end loop	**end loop**

Modeling variables d0, d1

- each variable: instance of a same process D
- read and write: RdV on events R and 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]

Modeling variable t

- variable t: instance of a process T
- read and write: RdV on events R and 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]

Modeling processes P0 and P1

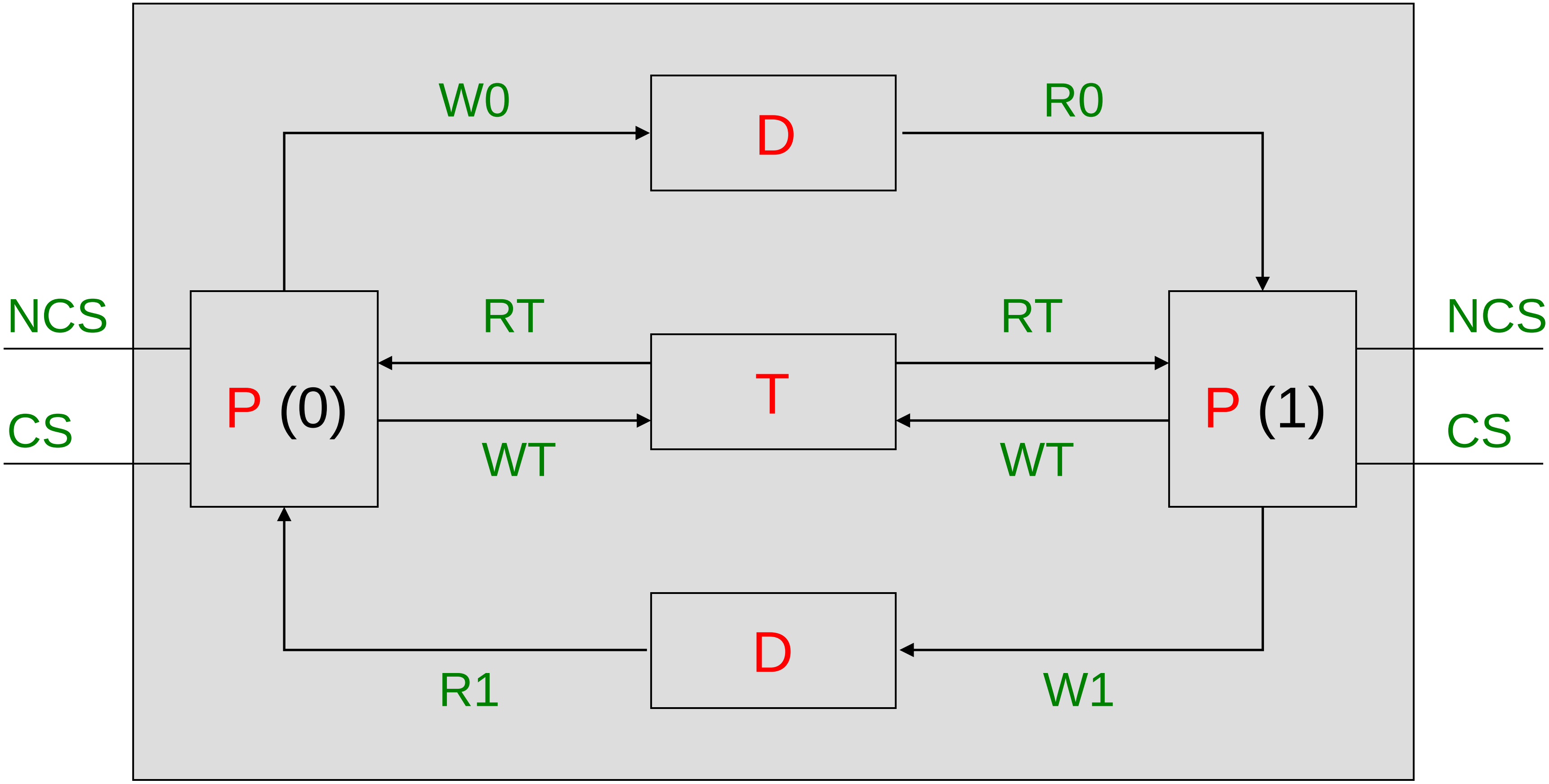
- instances of a same LNT process P
- process index: parameter of P
  - P0 ≡ P [W0, R1, RT, WT, NCS, CS] (0)
  - P1 ≡ P [W1, R0, RT, WT, NCS, CS] (1)

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

```
process P [Wm, Rn : 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);
               Wm (false);
               break wait
            end if
         end loop
      end loop
   end var
end process
```

System architecture



System architecture (cont'd)

```
process Main [NCS : none, CS : nat] is
   hide R0, W0, R1, W1 : bool, RT, WT : nat in
      par R0, W0, R1, W1, RT, WT in
         par      P [W0, R1, RT, WT, NCS, CS] (0)
         ||       P [W1, R0, RT, WT, NCS, CS] (1)
         end par
      ||
         par      T [RT, WT]
         ||       D [R0, W0]
         ||       D [R1, W1]
         end par
      end par
   end hide
end process
```

Alternative

```
process Main [NCS : none, CS : nat] is
  hide R0, W0, R1, W1 : bool, RT, WT : nat in
      par RT, WT in
            T [RT, WT]
      ||
            par
                  W0, R1 → P [W0, R1, RT, WT, NCS, CS] (0)
            ||    W1, R0 → P [W1, R0, RT, WT, NCS, CS] (1)
            ||    R0, W0 → D [R0, W0]
            ||    R1, W1 → D [R1, W1]
      end par
    end par
  end hide
end process
```

Remarks

- LNT is more concise than CA: process parameterisation, value exchange, etc.
- In general, there exist several ways to express the parallel composition of the system components (processes)
- For cyclic behaviours: choice between iterative (**loop**) and recursive (process call) styles

Top-down specification
(by successive refinements)

- identify the LNT processes that run in parallel (process MAIN)
- identify the visible and hidden events
- identify the interprocess synchronisations
- identify the initial values of each process parameters
- describe each process behaviour (usually sequential)

APPENDIX: OPERATIONAL SEMANTICS OF THE LNT LANGUAGE (IN FRENCH)

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 :

  $\rho + \rho' = \rho$ 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 :

  $\rho' - \rho =$ 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 $\rightarrow_e$ 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 : substitutions)
- **Exemple**

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

Sémantique du filtrage de motifs

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

Sémantique de « stop »

aucune règle sémantique associée à cet opérateur **stop** (inaction sans continuation)

stop ne permet de dériver aucune action

Cet opérateur n'existe pas dans la partie données

Sémantique de « null »

- Aucun effet, avec continuation : **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

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'

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

Communication sur une porte (2/2)

$$\frac{\rho_0 = \rho \qquad (\forall i \in 1..n)\ \{\ O_i \# v_i\ \}\ \rho_{i\text{-}1} \rightarrow_{\text{o}} \rho_i \qquad \rho' = \rho_n \qquad \{\ V\ \}\ \rho' \rightarrow_{\text{e}} \text{true}}{\{\ G\ (O_1, \ldots, O_n)\ \textbf{where}\ V\ \}\ \rho \xrightarrow{G\ (v_1, \ldots, v_n)}_{\text{b}} \{\ \textbf{null}\ \}\ \rho'}$$

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, ..., O_n) **where** false ≈ **stop**
- si plusieurs valeurs satisfont les conditions, il y a un choix non-déterministe entre ces valeurs

Sémantique de « ; »

$$B_1;\ B_2$$

- **Communication dans *B_1***

$$\frac{\{\ B_1\ \}\ \rho\ -G\ (v_1,\ \ldots,\ v_n)\!\rightarrow_{\mathrm{b}}\ \{\ B_1'\ \}\ \rho'}{\{\ B_1;\ B_2\ \}\ \rho\ -G\ (v_1,\ \ldots,\ v_n)\!\rightarrow_{\mathrm{b}}\ \{\ B_1';\ B_2\ \}\ \rho'}$$

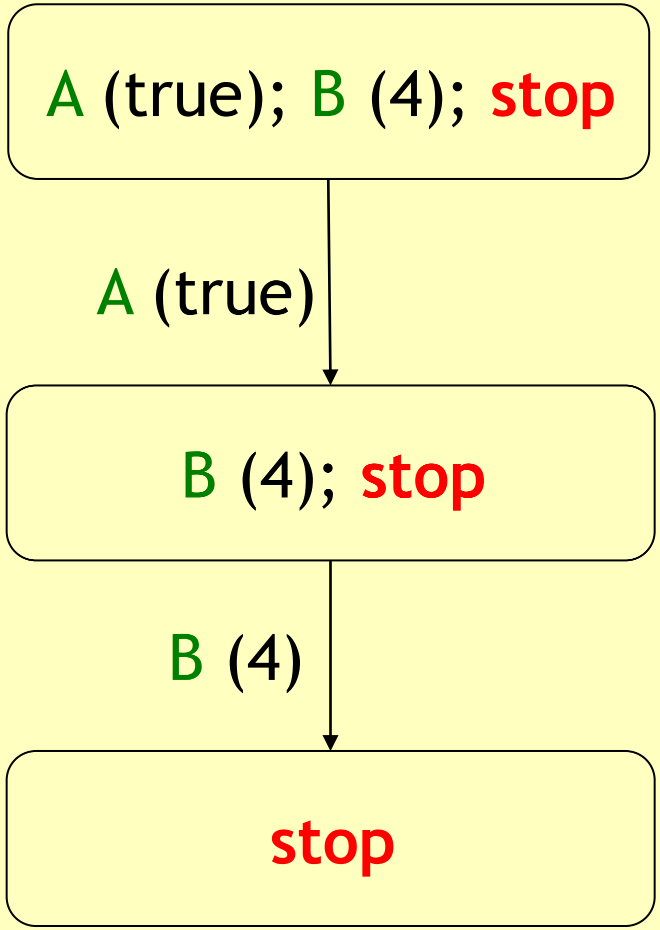
- **Terminaison de *B_1***

$$\frac{\{\ B_1\ \}\ \rho\ -\mathbf{exit}\!\rightarrow_{\mathrm{b}}\ \{\ B_1'\ \}\ \rho' \qquad \{\ B_2\ \}\ \rho'\ -l\!\rightarrow_{\mathrm{b}}\ \{\ B_2'\ \}\ \rho''}{\{\ B_1;\ B_2\ \}\ \rho\ -l\!\rightarrow_{\mathrm{b}}\ \{\ B_2'\ \}\ \rho''}$$

Exemples

Composition séquentielle :

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



Sémantique de « case »

$$\frac{\{\, V \,\}\, \rho \rightarrow_{\mathrm{e}} v \qquad j \in 1..m \qquad (\forall i \in 1..j\text{-}1)\ \{\, P_i \# v \,\}\, \rho \rightarrow_{\mathrm{p}} \textbf{fail} \qquad \{P_j \# v\} \rightarrow_{\mathrm{p}} \rho_j \qquad \{\, B_j \,\}\, \rho_j \xrightarrow{\ \iota\ }_b \{\, B_j' \,\}\, \rho_j'}{\{\, \textbf{case}\ V\ \textbf{in}\ P_1 \text{ -> } B_1 \mid \ldots \mid P_m \text{ -> } B_m\ \textbf{end case} \,\}\, \rho \xrightarrow{\ \iota\ }_b \{\, B_j' \,\}\, \rho_j'}$$

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

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

$\equiv$ **case** V **in** true -> B_1 | false -> B_2 **end case**

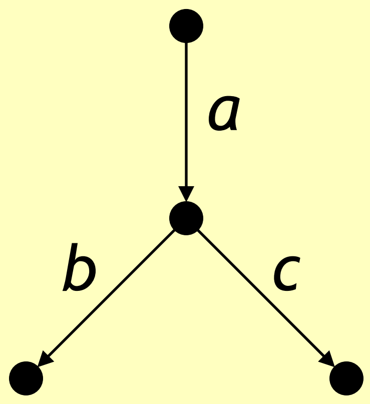
Opérateur « select »

- 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 :

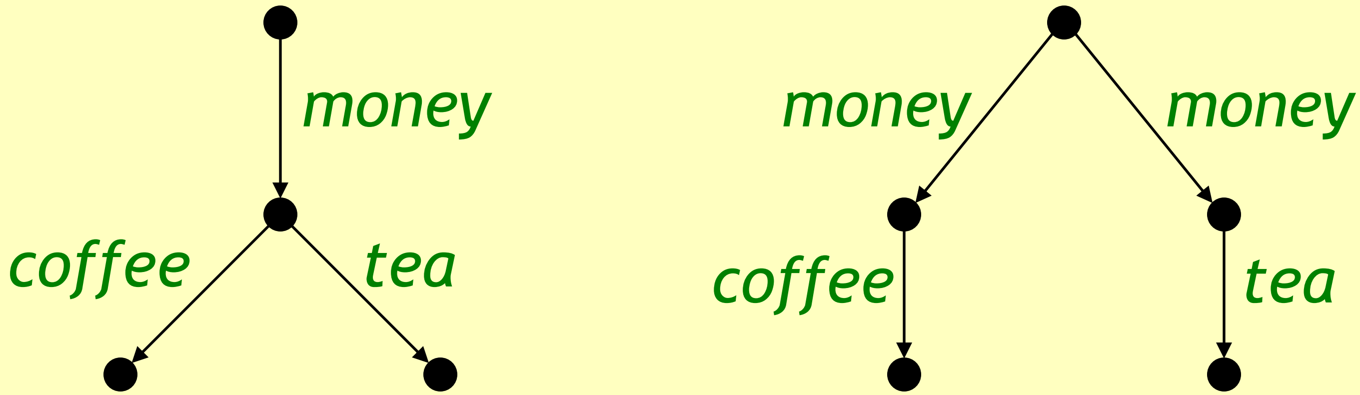
- n'existe pas dans la partie données
- après le choix, tous les autres comportements disparaîssent (on s'est engagé dans une branche du choix)

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)

Variables

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

  var $X_1{:}T_1, \ldots, X_n{:}T_n$ **in** B **end var**

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

Sémantique de « := »

- **Affectation déterministe**

$$\frac{\{V\}\ \rho \rightarrow_{\mathrm{e}} v}{\{X := V\}\ \rho \xrightarrow{\textbf{exit}}_{\mathrm{b}} \{\textbf{stop}\}\ \rho + [X := v]}$$

Rem : l'affectation de tableau est omise par simplicité

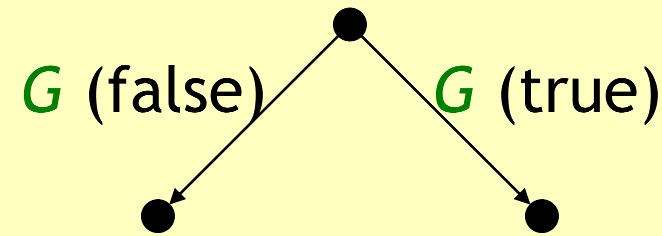
- **Affectation non-déterministe**

$$\frac{v \in T \qquad \rho' = \rho + [X := v] \qquad \{V\}\ \rho' \rightarrow_{\mathrm{e}} \text{true}}{\{X := \textbf{any}\ T\ \textbf{where}\ V\}\ \rho \xrightarrow{\textbf{exit}}_{\mathrm{b}} \{\textbf{stop}\}\ \rho'}$$

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

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**

Appel de processus

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

B

end process

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

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 **in out**) ; voir la doc
- L'appel de processus est interdit dans la partie données

Sémantique de « while »

$$\frac{\{V\}\ \rho \rightarrow_{\mathrm{e}} \text{true} \qquad \{\ B;\ \textbf{while}\ V\ \textbf{loop}\ B\ \textbf{end loop}\ \}\ \rho \xrightarrow{l}_{\mathrm{b}} \{\ B'\ \}\ \rho'}{\{\textbf{while}\ V\ \textbf{loop}\ B\ \textbf{end loop}\}\ \rho \xrightarrow{l}_{\mathrm{b}} \{\ B'\ \}\ \rho'}$$

$$\frac{\{\ V\ \}\ \rho \rightarrow_{\mathrm{e}} \text{false}}{\{\ \textbf{while}\ V\ \textbf{loop}\ B\ \textbf{end loop}\ \}\ \rho \xrightarrow{\textbf{exit}}_{\mathrm{b}} \{\ \textbf{stop}\ \}\ \rho}$$

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
```

Sémantique de « par » (1/2)

N'existe pas dans la partie données

Défini par 3 règles SOS

1. Entrelacement

$$\frac{i \in 0..m \quad \{\, B_i \,\}\, \rho \xrightarrow{G\ (v_1,\ \ldots,\ v_n)}_{\mathrm{b}} \{\, B_i' \,\}\, \rho' \qquad G \notin \underline{G}}{\{\, \textbf{par}\ \underline{G}\ \textbf{in}\ B_0 \ ||\ \ldots\ B_i\ \ldots\ ||\ B_m\ \textbf{end par} \,\}\, \rho \xrightarrow{G\ (v_1,\ \ldots,\ v_n)}_{\mathrm{b}} \{\, \textbf{par}\ \underline{G}\ \textbf{in}\ B_0 \ ||\ \ldots\ B_i'\ \ldots\ ||\ B_m\ \textbf{end par} \,\}\, \rho'}$$

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

Sémantique de « par » (2/2)

2. Synchronisation

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

3. Terminaison synchronisée

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

Sémantique de « hide »

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

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

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

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