PRESENTATION DU LANGAGE LOTOS[1]

Hubert Garavel

31 août 1993

[1](annexes A et B de la thèse de l'auteur)

Table des matières

Annexe A

Présentation du langage LOTOS : les structures de données

Comme la plupart des langages de programmation, LOTOS permet de définir et de manipuler des *structures de données*. Mais, à la différence des langages classiques, LOTOS utilise le formalisme des *types abstraits algébriques* (*abstract data types, ADT*), en s'inspirant largement du langage ACTONE [**?**] [**?**]. Dans une large mesure, le choix des types abstraits pour la description des structures de données est conforme aux objectifs d'un langage de spécification :

- les spécifications algébriques constituent un modèle mathématique exprimant les propriétés que doit vérifier toute réalisation, sans imposer de contraintes d'implémentation superflues
- les propriétés des données et des opérations sont complètement décrites. En choisissant les types abstraits, LOTOS évite les difficultés rencontrées en ESTELLE où les données sont spécifiées au moyen des types du langage PASCAL [**?**], ce qui pose des problèmes bien connus (combien vaut $-7 \bmod 3$?)

Cette annexe présente succinctement les principales caractéristiques de la description des données en LOTOS. Son propos n'est pas de se substituer à la définition formelle [**?**] mais d'aider à sa compréhension. Pour une étude approfondie des types abstraits, [**?**] constitue une référence.

A.1 Présentation des types abstraits

Les définitions suivantes sont nécessaires à la compréhension des types abstraits, tels qu'ils figurent en LOTOS :

sorte : nom donné à un domaine de valeurs. Par exemple `BOOL` dénote la sorte des valeurs booléennes et `NAT` celle des entiers naturels. L'utilisateur n'a pas à spécifier explicitement le domaine des sortes qu'il déclare

domaine (*data-carrier***) :** ensemble des valeurs d'une sorte

opérateur, opération : nom donné à une fonction qui à n arguments ($n \geq 0$) fait correspondre un résultat. Par exemple `false`, `true`, `not`, `and`, `or`, "`+`", "`*`", ... sont des opérations

arité : nombre d'arguments d'un opérateur. Par exemple, l'arité de `true` est 0, celle de `not` est 1, celle de "`+`" est 2

constante : opérateur d'arité nulle

profil : sortes des arguments et du résultat d'un opérateur. Par exemple le profil de `not` est `BOOL` → `BOOL` et celui de "`+`" est `NAT` × `NAT` → `NAT`

surcharge (*overloading*) : possibilité de définir plusieurs opérateurs possédant le même nom et des profils différents. Par exemple, l'opérateur d'égalité `eq` est surchargé puisqu'il peut avoir plusieurs profils : `NAT` × `NAT` → `BOOL` et `BOOL` × `BOOL` → `BOOL`

équation : égalité servant à définir la sémantique d'un opérateur. En LOTOS la sémantique des sortes et des opérateurs est exclusivement décrite par des équations. Par exemple, l'équation `P or Q = not (not (P) and not (Q))` exprime une relation qui peut constituer la définition de `or` à partir de `not` et `and`

type : nom donné à un ensemble de sortes, d'opérations et d'équations. On réunit dans un type des éléments qui décrivent un même concept : le type `BOOLEAN`, le type `NATURALNUMBER`

signature : ensemble des sortes et des opérations d'un type. Par exemple la signature du type `BOOLEAN` comprend, entre autres, `BOOL`, `false`, `true`, `not`, `and`, `or`. Celle du type `NATURALNUMBER` comprend `NAT`, "`+`", "`*`", ...

présentation : ensemble des sortes, des opérations et des équations d'un type

variable : nom donné à une valeur. LOTOS est un langage fonctionnel fortement typé : chaque variable ne peut prendre qu'une seule valeur d'une sorte déterminée par sa déclaration

expression de valeur, terme : expression construite à partir de variables et d'opérations. Par exemple, si `P` et `Q` sont des variables de sorte `BOOL` alors `P`, `not (P)`, `P and Q`, `not (P and Q)`, ... sont des termes

Les arguments des opérateurs unaires doivent être parenthésés (ainsi `not P` est syntaxiquement incorrect). Les opérateurs binaires peuvent être préfixés ou infixés. La surcharge d'opérateurs est autorisée, sous certaines conditions qui visent à interdire les ambiguïtés

congruence : deux termes sont congrus si l'on peut, en utilisant les propriétés et les transformations indiquées par les équations, démontrer qu'ils sont syntaxiquement identiques. Par exemple `true or false` et `false or true` sont congrus

algèbre quotient : ensemble quotient de l'ensemble des termes par la relation de congruence. Les valeurs sont des éléments de l'algèbre quotient, c'est-à-dire des classes d'équivalence. Par exemple la valeur *true* représente l'ensemble de termes : {`true, not (false), true or false`,...}

A.2 Eléments lexicographiques et syntaxiques

En LOTOS les lettres majuscules et minuscules sont identiques. Les identificateurs sont composés de lettres, de chiffres et du caractère "`_`" suivant les règles usuelles. Pour les noms d'opérateurs LOTOS permet l'utilisation de caractères spéciaux : l'utilisateur peut ainsi créer des opérateurs baptisés "`+`", "`<=>`", "`//`", ... Il est toutefois interdit de redéfinir les mots-réservés du langage : "`=`", "`->`", "`=>`", ...

Identificateurs

Chaque classe d'identificateurs est dénotée par un symbole non-terminal défini comme suit :

- S : sortes
- F : opérateurs
- T : types
- X : variables

Déclarations de variables

La construction suivante déclare un ensemble de variables $X_0, \ldots X_n$ ayant la même sorte S :

$$X_0, \ldots X_n : S$$

Pour alléger les notations, on emploie l'abréviation $\widehat{X}$ qui dénote une liste non vide de variables $X_0, \ldots X_n$.

Expressions de valeur

Les expressions de valeurs, dénotées par le non-terminal V, sont définies par les règles suivantes :

$$\begin{array}{rcl} V & \equiv & X \\ & | & F\ [(V_0, \ldots\ V_n)] \\ & | & V_1\ F\ V_2 \\ & | & V_0\ \textbf{of}\ S \end{array}$$

Les trois premières règles expriment qu'une expression de valeur peut être soit une variable, soit une opération préfixée (d'arité quelconque) ou infixée (d'arité 2). La quatrième règle introduit la notation "**of** S" qui permet de résoudre les ambiguïtés liées aux surcharges d'opérateurs en précisant que l'expression de valeur V_0 a pour sorte S.

Opérations

La construction suivante déclare un ensemble d'opérateurs $F_0, \ldots F_m$ ayant le même profil $S_1, \ldots S_n \rightarrow S$:

$$opns \quad \equiv \quad F_0, \ldots F_m \ \texttt{:}\ S_1, \ldots S_n \ \texttt{->}\ S$$

Pour spécifier qu'un opérateur binaire F est infixé il suffit, dans la déclaration *opns*, d'entourer F de deux caractères "`_`" : "`_+_`", "`_<=>_`", "`_//_`", ...

Equations

Le non-terminal *seq* dénote une équation "simple" dont les membres sont deux expressions de valeurs V_1 et V_2 :

$$seq \quad \equiv \quad V_1\texttt{=}V_2$$

Le non-terminal *peq* dénote une "prémisse" formée soit d'une équation simple soit d'une expression de valeur :

$$\begin{array}{rcl} peq & \equiv & seq \\ & | & V \end{array}$$

Le non-terminal *meq* dénote une équation "moyenne" formée d'une équation simple *seq*, éventuellement précédée d'un ensemble de prémisses (on parle alors d'*équation conditionnelle*) :

$$meq \equiv [peq_0, \ldots\ peq_n \texttt{=>}]\ seq$$

Le non-terminal *ceq* dénote une équation "complexe" formée d'un ensemble d'équations moyennes $meq_0, \ldots\ meq_n$ dont les membres, gauches et droits, ont tous la même sorte S. Les membres des équations sont des termes du 1er ordre pouvant contenir des variables $\widehat{X_0}, \ldots\ \widehat{X_m}$ quantifiées universellement :

$$ceq \equiv \textbf{ofsort}\ S\ [\textbf{forall}\ \widehat{X_0}:S_0, \ldots\ \widehat{X_m}:S_m]\ meq_0, \ldots\ meq_n$$

Le non-terminal *eqns* dénote un ensemble d'équations complexes $ceq_0, \ldots\ ceq_n$ dans lesquelles peuvent figurer des variables $\widehat{X_0}, \ldots\ \widehat{X_m}$ quantifiées universellement :

$$eqns \equiv [\textbf{forall}\ \widehat{X_0}:S_0, \ldots\ \widehat{X_m}:S_m]\ ceq_0, \ldots\ ceq_n$$

A.3 Types importés

Un programme LOTOS peut importer des types prédéfinis. La construction suivante déclare les types $T_0, \ldots\ T_n$ dont la définition doit être recherchée dans la bibliothèque LOTOS :

library $T_0, \ldots\ T_n$
endlib

La définition de LOTOS n'impose aucune contrainte sur la façon dont cette bibliothèque doit être réalisée. En revanche elle fournit une liste de types prédéfinis qui constituent la *bibliothèque standard* du langage [**?**, annexe A].

A.4 Types élémentaires

La construction suivante déclare un type élémentaire T en lui associant sa présentation, composée de sortes, d'opérations et d'équations :

type T **is**

type T **is** $T_0, \ldots\ T_n$
[**sorts** $S_0, \ldots\ S_p$]
[**opns** $opns_0, \ldots\ opns_q$]
[**eqns** *eqns*]
endtype

Exemple A-1
L'exemple suivant est extrait de la bibliothèque standard du langage LOTOS [**?**, annexe A.4]. Il définit le type abstrait `BOOLEAN` qui exporte une sorte `BOOL`, deux constantes `false` et `true`, un opérateur unaire `not` et sept opérateurs binaires infixés `and`, `or`, ... En appliquant les équations, n'importe quel terme peut être évalué soit à `false` soit à `true` :

```
type BOOLEAN is
   sorts BOOL
   opns true, false : -> BOOL
        not : BOOL -> BOOL
        _and_, _or_, _xor_, _implies_, _iff_ : BOOL, BOOL -> BOOL
        _eq_, _ne_ : BOOL, BOOL -> BOOL
   eqns forall X, Y :BOOL
        ofsort BOOL
           not (true) = false;
           not (false) = true;

           X and true = X;
           X and false = false;

           X or true = true;
           X or false = X;

           X xor Y = (X and not (Y)) or (Y and not (X));
           X implies Y = Y or not (X);
           X iff Y = (X implies Y) and (Y implies X);
           X eq Y = X iff Y;
           X ne Y = X xor Y
endtype
```

■

A.5 Types combinés

La combinaison (*combination*) permet de réutiliser des types déjà existants pour définir de nouveaux types. La construction suivante déclare un type T obtenu par combinaison d'un ensemble de types $T_0, \ldots\ T_n$ définis auparavant. Le type T est défini par sa présentation, de la même manière qu'un type élémentaire :

type T **is** $T_0, \ldots\ T_n$
[**sorts** $S_0, \ldots\ S_p$]
[**opns** $opns_0, \ldots\ opns_q$]
[**eqns** *eqns*]
endtype

Les sortes et les opérations qui font partie de la signature de $T_0, \ldots T_n$ peuvent être utilisées dans la présentation de T. On dit que le type T *hérite*[1] des types $T_0, \ldots T_n$.

Exemple A-2
L'exemple suivant est une variante de la définition abstraite des nombres naturels donnée dans la bibliothèque standard [**?**, annexe A.6.1.1]. Il définit le type `NATURAL` qui exporte la sorte `NAT`, la constante `0` et l'opérateur unaire `SUCC` ; chaque nombre possède une représentation canonique en base 1 définie comme suit :

$$\begin{array}{lcl} 0 & = & \texttt{0} \\ 1 & = & \texttt{SUCC (0)} \\ 2 & = & \texttt{SUCC (SUCC (0))} \\ & \ldots & \end{array}$$

Deux opérations plus complexes, l'addition (opérateur binaire infixé "`+`") et la multiplication (opérateur binaire infixé "`*`"), sont définies en fonction des opérateurs `0` et `SUCC` au moyen d'équations.

Le type `NATURAL` contient aussi les relations de comparaison entre nombres naturels, c'est-à-dire six opérateurs binaires infixés : `eq` (*equal*), `ne` (*not equal*), `lt` (*less than*), `le` (*less or equal*), `gt` (*greater than*) et `ge` (*greater or equal*).

Le résultat de ces opérateurs est de sorte `BOOL`. La définition du type `NATURAL` utilise celle du type `BOOLEAN` puisqu'elle importe la sorte `BOOL`.

```
type NATURAL is BOOLEAN
   sorts NAT
   opns 0 : -> NAT
        SUCC : NAT -> NAT
        _+_, _*_ : NAT, NAT -> NAT
        _eq_, _ne_, _lt_, _le_, _gt_, _ge_ : NAT, NAT -> BOOL
   eqns forall M, N:NAT
        ofsort NAT
           M + 0 = M;
           M + SUCC (N) = SUCC (M + N);

           M * 0 = 0;
           M * SUCC (N) = M + (M * N)
        ofsort BOOL
           0 eq 0 = true;
           0 eq SUCC (M) = false;
           SUCC (M) eq 0 = false;
           SUCC (M) eq SUCC (N) = M eq N;

           0 lt 0 = false;
           0 lt SUCC (M) = true;
           SUCC (M) lt 0 = false;
           SUCC (M) lt SUCC (N) = M lt N;

           M ne N = not (M eq N);
           M le N = (M lt N) or (M eq N);
           M gt N = not (M le N);
           M ge N = not (M lt N)
endtype
```

[1] il s'agit d'un héritage multiple

■

Exemple A-3
Il est possible de spécifier des structures de données plus complexes que les entiers et les booléens. L'exemple suivant montre comment décrire les listes LISP en LOTOS. On suppose que les éléments des listes (*atomes*) sont des nombres naturels. Le type `NATURAL_LIST` exporte une sorte `LIST` et les cinq opérations primitives de LISP : `NIL`, `CONS`, `CAR`, `CDR` et `ATOM`. Toute liste peut s'exprimer en fonction de `NIL` et `CONS` de la manière suivante :

$$\begin{aligned} () &= \texttt{NIL} \\ (n_1) &= \texttt{CONS (}n_1\texttt{, NIL)} \\ (n_1, n_2) &= \texttt{CONS (}n_1\texttt{, CONS (}n_2\texttt{, NIL))} \\ &\ldots \end{aligned}$$

Les équations donnent aux fonctions `CAR`, `CDR` et `ATOM` leur sémantique usuelle. Les termes `CAR (NIL)` et `CDR (NIL)` sont indéfinis, donc irréductibles.

Ces opérations ont des paramètres et des résultats de sorte `NAT` et `BOOL`. La définition du type `NATURAL_LIST` utilise donc celles des types `NATURAL` et `BOOLEAN`.

```
type NATURAL_LIST is NATURAL, BOOLEAN
   sorts LIST
   opns NIL : -> LIST
        CONS : NAT, LIST -> LIST
        CAR : LIST -> NAT
        CDR : LIST -> LIST
        ATOM : LIST -> BOOL
   eqns forall N:NAT, L:LIST
        ofsort LIST
           CAR (CONS (N, L)) = N;
           CDR (CONS (N, L)) = L
        ofsort BOOL
           ATOM (NIL) = true;
           ATOM (CONS (N, L)) = false
endtype
```

■

La combinaison peut aussi servir à *enrichir* un type existant en le complétant par de nouvelles sortes et de nouveaux opérateurs.

Exemple A-4
Il est possible d'enrichir le type `NATURAL_LIST` défini dans l'exemple A-3 (p. 9) en lui ajoutant les relations d'égalité et d'inégalité sur les listes (opérateurs binaires infixés `eq` et `ne`). On obtient ainsi le type `ENRICHED_NATURAL_LIST` :

```
type ENRICHED_NATURAL_LIST is NATURAL_LIST
   opns _eq_, _ne_ : LIST, LIST -> BOOL
   eqns forall N, N1, N2:NAT, L, L1, L2:LIST
        ofsort BOOL
           NIL eq NIL = true;
           NIL eq CONS (N, L) = false;
           CONS (N, L) eq NIL = false;
           N1 eq N2 =>
              CONS (N1, L1) eq CONS (N2, L2) = (L1 eq L2);
           N1 ne N2 =>
              CONS (N1, L1) eq CONS (N2, L2) = false;

           L1 ne L2 = not (L1 eq L2)
endtype
```

L'emploi d'équations conditionnelles pour la définition de l'égalité n'est pas indispensable ; on aurait pu remplacer les deux équations conditionnelles par l'équation suivante :

```
CONS (N1, L1) eq CONS (N2, L2) = (N1 eq N2) and (L1 eq L2)
```

■

A.6 Types paramétrés

LOTOS permet la définition de types paramétrés par des *sortes formelles* et des *opérateurs formels*.

Pour cela on commence par définir un *type formel* T dont la présentation contient les sortes formelles et les opérations formelles servant de paramètres. La présentation de T peut également posséder des *équations formelles* qui, en spécifiant les propriétés des opérateurs formels, imposent des contraintes supplémentaires :

type T **is** $T_1, \ldots\ T_n$
  [**formalsorts** $S_0, \ldots\ S_p$]
  [**formalopns** $opns_0, \ldots\ opns_q$]
  [**formaleqns** *eqns*]
endtype

Les sortes et les opérations utilisées dans $opns_0, \ldots\ opns_q$ et *eqns* doivent être formelles.

Exemple A-5
Il est possible de généraliser le type ENRICHED_NATURAL_LIST défini dans l'exemple A-4 (p. 9) pour manipuler des listes dont les éléments sont de sorte quelconque — et non plus seulement de sorte NAT. On suppose néanmoins que tous les atomes ont la même sorte.

Pour cela il faut modifier la définition de ENRICHED_NATURAL_LIST en remplaçant toutes les occurrences de la sorte NAT par une sorte formelle appelée, par exemple, ITEM. Ce n'est pas suffisant : il faut également remplacer les opérations :

```
_eq_, _ne_ :  NAT, NAT -> BOOL
```

dont le profil dépend de la sorte NAT par les opérations formelles correspondantes sur la sorte ITEM. Par commodité on conservera pour ces opérations formelles les noms eq et ne :

```
_eq_, _ne_ :  ITEM, ITEM -> BOOL
```

On peut spécifier certaines propriétés que doivent vérifier les opérations formelles, comme par exemple :

```
(forall N1, N2:ITEM) N1 ne N2 = not (N1 eq N2)
```

Au niveau des types, cette transformation revient à remplacer le type `NATURAL` par un type formel `FORMAL_ITEM` défini comme suit :

```
type FORMAL_ITEM is BOOLEAN
   formalsorts ITEM
   formalopns _eq_, _ne_ : ITEM, ITEM -> BOOL
   formaleqns forall N1, N2:ITEM
              ofsort BOOL
                 N1 ne N2 = not (N1 eq N2)
endtype
```

■

La définition d'un type paramétré s'effectue par combinaison avec un ou plusieurs types contenant des sortes (*resp.* opérations) formelles.

Exemple A-6
On peut alors définir le type abstrait `GENERIC_LIST` décrivant les listes LISP paramétrées :

```
type GENERIC_LIST is FORMAL_ITEM, BOOLEAN
   sorts LIST
   opns NIL : -> LIST
        CONS : ITEM, LIST -> LIST
        CAR : LIST -> ITEM
        CDR : LIST -> LIST
        ATOM : LIST -> BOOL
        _eq_, _ne_ : LIST, LIST -> BOOL
   eqns forall N, N1, N2:ITEM, L, L1, L2:LIST
        ofsort LIST
           CAR (CONS (N, L)) = N;
           CDR (CONS (N, L)) = L
        ofsort BOOL
           ATOM (NIL) = true;
           ATOM (CONS (N, L)) = false;

           NIL eq NIL = true;
           NIL eq CONS (N, L) = false;
           CONS (N, L) eq NIL = false;
           N1 eq N2 =>
              CONS (N1, L1) eq CONS (N2, L2) = (L1 eq L2);
           N1 ne N2 =>
              CONS (N1, L1) eq CONS (N2, L2) = false;

           L1 ne L2 = not (L1 eq L2)
endtype
```

■

Remarque A-1
En fait, il n'y a pas de distinction stricte entre types élémentaires et types formels puisqu'un même type peut simultanément contenir des sortes (*resp.* opérations, équations) formelles et non formelles. La construction générale permettant la déclaration d'un type est :

type T **is** $T_1, \ldots\ T_n$
[**formalsorts** $S_0, \ldots\ S_p$]

[**formalopns** $opns_0, \ldots\ opns_q$]
[**formaleqns** $eqns$]
[**sorts** $S'_0, \ldots\ S'_{p'}$]
[**opns** $opns'_0, \ldots\ opns'_{q'}$]
[**eqns** $eqns'$]
endtype

Il n'est donc pas indispensable de définir séparément les types `FORMAL_ITEM` et `GENERIC_LIST` : les clauses "**formalsorts**", "**formalopns**" et "**formaleqns**" figurant dans `FORMAL_ITEM` peuvent être directement insérées dans `GENERIC_LIST`. ■

A.7 Types instanciés

La construction suivante déclare un type T obtenu par instanciation (*actualization*) d'un type T' par les sortes formelles $S'_0, \ldots\ S'_p$ et les opérations formelles $F'_0, \ldots\ F'_q$. Pour instancier T' on substitue aux paramètres formels $S'_0, \ldots\ S'_p$, $F'_0, \ldots\ F'_q$ des paramètres effectifs, respectivement $S_0, \ldots\ S_p$, $F_0, \ldots\ F_q$. On indique l'ensemble de types $T_0, \ldots\ T_n$ dont les signatures contiennent les sortes et les opérations effectives :

type T **is** T' **actualizedby** $T_0, \ldots\ T_n$ **using**
[**sortnames** S_0 **for** $S'_0, \ldots\ S_p$ **for** S'_p]
[**opnnames** F_0 **for** $F'_0, \ldots\ F_q$ **for** F'_q]
endtype

L'instanciation peut être partielle, c'est à dire que certaines sortes ou opérations formelles de T' peuvent ne pas être instanciées. Dans ce cas T est, lui aussi, un type paramétré.

Exemple A-7
Pour définir des listes dont les atomes sont booléens, il suffit d'instancier le type `GENERIC_LIST` défini dans l'exemple A-5 (p. 10) en établissant la correspondance suivante :

```
FORMAL_ITEM  ⟶  BOOLEAN
       ITEM  ⟶  BOOL
         eq  ⟶  iff
         ne  ⟶  xor
```

En LOTOS, cette instanciation s'écrit ainsi :

```
type BOOLEAN_LIST is GENERIC_LIST actualizedby BOOLEAN using
   sortnames BOOL for ITEM
   opnnames iff for eq
            xor for ne
endtype
```

■

A.8 Types renommés

La construction suivante déclare un type T obtenu par renommage (*renaming*) d'un type T'. La présentation de T est identique à celle de T' dans laquelle les sortes S'_0, ... S'_p et les opérations F'_0, ... F'_q sont respectivement renommées en S_0, ... S_p, F_0, ... F_q :

type T **is** T' **renamedby**
  [**sortnames** S_0 **for** S'_0,... S_p **for** S'_p]
  [**opnnames** F_0 **for** F'_0,... F_q **for** F'_q]
endtype

Le renommage peut être utilisé pour changer les notations des sortes et des opérations d'un type existant.

Exemple A-8
Il est possible de donner aux opérateurs de comparaison du type `NATURAL` défini dans l'exemple A-2 (p. 8) une lexicographie plus familière (il n'est pas possible de définir le symbole "=" qui est un mot réservé) :

```
type RENAMED_NATURAL is NATURAL renamedby
   opnnames == for eq
               <> for ne
               <  for lt
               <= for le
               >  for gt
               >= for ge
endtype
```

■

Le renommage est aussi utilisé pour construire un nouveau type ayant la même structure qu'un type existant, mais destiné à modéliser d'autres objets que le type existant. On retrouve cette approche en ADA avec le mécanisme des types dérivés (**type** T **is new** T').

Exemple A-9
Mathématiquement parlant, les deux corps commutatifs $(\{false, true\}, xor, and)$ et $(\{0,1\}, +, .)$, où "+" dénote l'addition modulo 2 et "." la multiplication, sont isomorphes. En LOTOS il est possible de définir l'une de ces structures algébriques comme simple renommage de l'autre :

```
type BINARY is BOOLEAN renamedby
   sortnames BIN for BOOL
   opnnames 0 for false
            1 for true
            + for xor
            . for and
endtype
```

■

Annexe B

Présentation du langage LOTOS : les structures de contrôle

Après les structures de données, cette annexe présente, de manière simple et accessible, l'autre partie du langage LOTOS : la description du contrôle. On peut également consulter le *tutorial* LOTOS [**?**].

LOTOS est un langage parallèle qui s'inspire des *algèbres de processus*, notamment CCS [**?**] et TCSP [**?**] : le contrôle des programmes est décrit par des expressions algébriques appelées *comportements*. La synchronisation et la communication s'effectuent exclusivement par rendez-vous, sans partage de mémoire.

B.1 Eléments lexicographiques et syntaxiques

B.1.1 Expressions de comportement

On appelle *opérateurs de comportement* les structures de contrôle utilisées dans le langage : composition séquentielle, composition parallèle, ... Dans la suite ces opérateurs sont présentés l'un après l'autre.

On appelle *expression de comportement* (*behaviour expression*) — ou plus simplement *comportement* — un terme syntaxique obtenu par combinaison des opérateurs de comportement. Les expressions de comportement sont dénotées par le non-terminal B.

B.1.2 Expressions de valeur

On appelle *expression de valeur* (*value expression*) — ou plus simplement *valeur* — un terme algébrique construit à partir de variables et d'opérateurs. Les expressions de valeur ont été définies au chapitre précédent, où elles apparaissaient dans les équations algébriques ; elles sont aussi utilisées dans les expressions de comportement. Les expressions de valeur sont dénotées par le non-terminal V.

LOTOS est fortement typé : chaque valeur ne peut avoir qu'une seule sorte, qu'il est possible de déterminer statiquement.

Les sortes et les opérations qui sont utilisées dans les comportements LOTOS ne doivent pas être

formelles ; elles doivent appartenir à des types complètement instanciés.

Si S est une sorte, on note $domain(S)$ l'ensemble quotient de tous les termes de sorte S par la relation de congruence définie par les équations associées à S. On fait l'hypothèse que le domaine de chaque sorte n'est pas vide.

Si V_1 et V_2 sont deux valeurs de sorte S, on note $V_1 = V_2$ le fait que V_1 et V_2 soient congrues modulo la relation de congruence définie par les équations associées à S.

B.1.3 Variables

Une *variable* est un nom donné à une valeur. LOTOS est un langage fonctionnel : chaque variable est initialisée dès sa déclaration et sa valeur ne peut pas être modifiée.

B.1.4 Portes

En LOTOS, on appelle *porte* (*gate*) un canal de communication permettant la synchronisation par rendez-vous et l'échange de valeurs entre plusieurs tâches qui se déroulent en parallèle.

On note Γ l'ensemble de tous les identificateurs de portes, définis par l'utilisateur, qui figurent dans une spécification LOTOS. Deux portes spéciales sont prédéfinies, qui n'appartiennent pas à Γ :

- la porte invisible, notée "**i**"[2]. Cette porte peut apparaître dans les programmes LOTOS, mais uniquement dans le contexte d'un opérateur "**;**"
- la porte de terminaison, notée "δ". Cette porte ne peut jamais être employée explicitement dans un programme LOTOS mais elle est utilisée dans la définition sémantique du langage

B.1.5 Identificateurs

Chaque classe d'identificateurs est dénotée par un symbole non-terminal défini comme suit :

- G : portes
- P : processus
- X : variables
- T : types
- S : sortes
- F : opérations

On emploie en outre les abréviations suivantes :

- $\widehat{G}$: liste non vide de portes $G_0, \ldots\ G_n$
- $\widehat{X}$: liste non vide de variables $X_0, \ldots\ X_n$

On introduit à présent tous les opérateurs de contrôle du langage LOTOS. Un même exemple, celui d'un distributeur de boissons, est conservé tout au long de la présentation.

[2] qui correspond à la porte "τ" de CCS

B.2 Opérateur "stop"

La construction suivante :

stop

dénote un comportement inactif, qui ne propose aucun rendez-vous avec l'environnement ni aucune transition "**i**" interne.

B.3 Opérateur ";"

L'opérateur "`;`" permet de spécifier le rendez-vous. Si G est une porte et B_0 un comportement, la construction suivante :

$$G \ ; \ B_0$$

dénote le comportement qui propose un rendez-vous sur la porte G et, une fois qu'il a eu lieu, exécute B_0. La notation "`;`" a une signification séquentielle : on dit que le comportement B est *préfixé* par la porte G. Les termes *événement* et *interaction* seront utilisés comme synonymes de rendez-vous.

Exemple B-1
Le comportement suivant effectue une interaction `MONEY` (acquisition de pièces de monnaie) puis une interaction `TEA` (distribution d'une tasse de thé), après quoi il s'arrête :

```
MONEY;
   TEA;
      stop
```

En fait cet exemple décrit également le comportement d'un utilisateur qui, après avoir payé, reçoit une tasse de thé. ■

Cette forme simple de rendez-vous, qui ne comporte pas d'émission ni de réception de valeurs, ne permet que la *synchronisation pure*. Il existe une construction plus générale qui prend en compte l'échange de valeurs :

$$G \ O_0, \ldots \ O_n \ ; \ B_0$$

où $O_0, \ldots \ O_n$ sont des *offres*, définies comme suit :

$$\begin{array}{rcl} O & \equiv & \texttt{!}V \\ & | & \texttt{?}X_0, \ldots \ X_n\texttt{:}S \end{array}$$

Une offre de la forme "`!`V" correspond à l'émission sur la porte G de la valeur de l'expression V. Une offre de la forme "`?`$X_0, \ldots \ X_n$`:`S" correspond à la réception sur la porte G de $n+1$ valeurs $v_0, \ldots \ v_n$ de sorte S ; chacune de ces valeurs v_i est ensuite affectée à la variable X_i correspondante.

Le rendez-vous est bloquant aussi bien pour l'émission que la réception : l'exécution d'un comportement qui attend un rendez-vous est suspendue et ne reprend qu'après que le rendez-vous a eu lieu. Le rendez-vous LOTOS est absolument symétrique ; aucune distinction n'est faite entre émetteur et récepteur.

Un seul et même rendez-vous peut comporter plusieurs émissions et réceptions qui se déroulent simultanément. De plus, une même porte peut être successivement utilisée dans plusieurs rendez-vous, tantôt avec des émissions, tantôt avec des réceptions.

LOTOS permet de conditionner le rendez-vous par une *garde* qui est soit une expression booléenne "[V_0]", soit une équation simple "[V_1=V_2]". Le rendez-vous n'a pas lieu si la condition définie par la garde n'est pas satisfaite.

Exemple B-2
Le comportement suivant modélise un distributeur qui effectue successivement trois interactions :

- acquisition d'une somme d'argent (interaction `MONEY`) en dollars et en cents ; au moyen d'une garde on interdit le rendez-vous si cette somme est inférieure au prix attendu
- distribution d'une tasse de thé (interaction `TEA`)
- restitution de la monnaie (interaction `CHANGE`)

```
MONEY ?DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST];
   TEA;
      CHANGE !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
         stop
```

Les opérations comptables sont décrites à l'aide d'un type abstrait :

```
type CHANGE is NATURALNUMBER, BOOLEAN
   opns COST : -> NAT
        TOTAL : NAT, NAT -> NAT
        CHG_DOLLARS : NAT, NAT -> NAT
        CHG_CENTS : NAT, NAT -> NAT
   eqns forall DOLLARS, CENTS:NAT ofsort NAT
       COST = 25;   (* prix d'une boisson, exprime en cents *)
       TOTAL (DOLLARS, CENTS) = (100 * DOLLARS) + CENTS;
       CHG_DOLLARS (DOLLARS, CENTS) = (TOTAL (DOLLARS, CENTS) - COST) div 100;
       CHG_CENTS (DOLLARS, CENTS) = (TOTAL (DOLLARS, CENTS) - COST) mod 100;
endtype
```

■

On peut faire figurer la porte "**i**" à gauche de l'opérateur "**;**", mais elle ne doit comporter ni offre ni garde. Le préfixage par la porte "**i**" spécifie une évolution interne qui n'est jamais bloquante.

En résumé la syntaxe générale de l'opération de préfixage est donc :

$$\begin{array}{ll} & \mathbf{i}\ ;\ B_0 \\ | & G\ [O_0, \ldots\ O_n\ [\texttt{[}V_0\texttt{]}]]\ ;\ B_0 \\ | & G\ [O_0, \ldots\ O_n\ [\texttt{[}V_1\texttt{=}V_2\texttt{]}]]\ ;\ B_0 \end{array}$$

Les variables éventuellement définies dans les offres O_0, ... O_n ne sont visibles que dans V_0, V_1, V_2 et B_0.

B.4 Opérateur "[]"

L'opérateur "[]" permet de spécifier le choix non-déterministe. Si B_1 et B_2 sont deux comportements, la construction suivante :

$$B_1\ \texttt{[]}\ B_2$$

dénote le comportement qui peut exécuter soit B_1 soit B_2. La signification intuitive de cet opérateur est identique à celle de la barre carrée "[]" de Dijkstra.

Remarque B-1
Il n'est pas permis d'écrire en LOTOS des comportements de la forme :

$$(G_1 \texttt{ [] } G_2) \texttt{ ; } G_3 \texttt{ ; } \textbf{stop}$$

Il s'agit d'une erreur syntaxique car les opérandes de "[]" doivent être des comportements et non des portes ; de même l'opérande gauche de ";" doit être une porte et non un comportement. La manière correcte d'écrire le comportement ci-dessus est :

$$(G_1 \texttt{ ; } G_3 \texttt{ ; } \textbf{stop}) \texttt{ [] } (G_2 \texttt{ ; } G_3 \texttt{ ; } \textbf{stop})$$

■

Exemple B-3
Le comportement suivant modélise un distributeur qui, après avoir accepté le paiement (interaction `MONEY`) peut délivrer du thé, du café ou du chocolat chaud (interactions `TEA`, `COFFEE` et `CHOCOLATE`) :

```
MONEY;
   (
   TEA;
      stop
   []
   COFFEE;
      stop
   []
   CHOCOLATE;
      stop
   )
```

■

Le fait que le choix soit non-déterministe ne signifie pas que le distributeur décide arbitrairement de la boisson qu'il fournit. Le distributeur doit respecter les contraintes imposées par l'*environnement*, c'est-à-dire les rendez-vous proposés par les autres comportements avec lesquels il communique et se synchronise.

Dans l'exemple B-3 (p. 19) ce sont les utilisateurs du distributeur qui constituent cet environnement : après avoir payé (interaction `MONEY`), chaque consommateur sélectionne la boisson qu'il désire, par exemple en appuyant sur un bouton. S'il choisit le café, l'interaction `COFFEE` est imposée au distributeur.

Le non-déterminisme intervient effectivement lorsque les contraintes de l'environnement ne déterminent pas une possibilité unique. Pour reprendre l'exemple B-3 (p. 19), si le consommateur pouvait indiquer qu'il désire soit du café, soit du chocolat, le choix du distributeur (interaction `COFFEE` ou `CHOCOLATE`) serait imprévisible.

Il existe des comportements pour lesquels aucune interprétation déterministe ne peut être trouvée, quel que soit l'environnement. L'exemple le plus simple est de la forme :

$$(G \texttt{ ; } B_1) \texttt{ [] } (G \texttt{ ; } B_2)$$

Dans ce cas le choix entre l'exécution de B_1 ou de B_2 est purement arbitraire. La porte "**i**" introduit également le non-déterminisme. Quel que soit l'environnement, le comportement suivant :

$$B_1 \texttt{ [] } \textbf{i} \texttt{ ; } B_2$$

ne peut pas recevoir d'interprétation déterministe (sauf dans le cas où B_1 est égal à "**stop**"). En effet l'événement "**i**" est toujours possible car l'environnement n'a aucune influence sur lui. Pour

exprimer un choix non contrôlable par l'environnement, entre deux comportements B_1 et B_2, il faut écrire :

$$\mathbf{i}\ ;\ B_1\ \texttt{[]}\ \mathbf{i}\ ;\ B_2$$

L'opérateur "`[]`" est commutatif et associatif ; il admet "**stop**" comme élément neutre à gauche (*resp.* à droite) ; tout comportement est idempotent pour "`[]`". Une erreur fréquente consiste à croire que "`;`" est distributif sur "`[]`" et à écrire :

$$(G\ ;\ B_1)\ \texttt{[]}\ (G\ ;\ B_2)$$

au lieu de :

$$G\ ;\ (B_1\ \texttt{[]}\ B_2)$$

L'emploi de la première forme peut provoquer un blocage comme le montre l'exemple suivant.

Exemple B-4
Le distributeur de boissons présenté dans l'exemple B-3 (p. 19) ne doit pas être décrit ainsi :

```
MONEY;
   TEA;
      stop
[]
MONEY;
   COFFEE;
      stop
[]
MONEY;
   CHOCOLATE;
      stop
```

Au moment où le consommateur paie (interaction `MONEY`) le distributeur se trouve confronté à un choix non-déterministe, qu'il résout en sélectionnant arbitrairement une branche, au détriment des autres. S'il a choisi par exemple, la première il ne pourra plus délivrer que du thé ! En d'autres termes le choix proposé à l'utilisateur après paiement est restreint à une seule des trois interactions `TEA`, `COFFEE` et `CHOCOLATE`. ■

B.5 Opérateur "choice" sur les portes

Soit B_0 un comportement qui contient des occurrences d'utilisation d'une porte G. Soient G_1, ... G_n des portes et soient B_1, ... B_n les comportements définis de la manière suivante : B_i est obtenu à partir de B en remplaçant G par G_i. Pour exprimer le comportement :

$$B_1\ \texttt{[]}\ \ldots\ B_n$$

LOTOS permet d'utiliser une notation abrégée :

$$\textbf{choice}\ G\ \textbf{in}\ \texttt{[}G_1, \ldots\ G_n\texttt{]}\ \texttt{[]}\ B_0$$

La porte G sert d'indice à cette itération. Il n'est pas indispensable que les portes G_1, ... G_n soient deux à deux distinctes.

Exemple B-5
L'exemple B-3 (p. 19) peut être écrit de manière plus concise :

```
MONEY;
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         stop
   )
```

■

L'opérateur "**choice**" possède une forme plus générale permettant d'itérer sur plusieurs portes :

$$\textbf{choice } \widehat{G_0} \textbf{ in } [\widehat{G'_0}], \ldots \widehat{G_n} \textbf{ in } [\widehat{G'_n}] \ [] \ B_0$$

Les portes définies dans les listes $\widehat{G_0}$, ... $\widehat{G_n}$ ne sont visibles que dans B_0.

B.6 Opérateurs "||", "|||" et "|[...]|"

Les opérateurs qui ont été présentés jusqu'ici sont strictement *séquentiels* ; LOTOS comprend aussi des opérateurs *parallèles*. Si B_1 et B_2 sont deux comportements et G_0, ... G_n une liste de portes, la construction suivante :

$$B_1 \ |[G_0, \ldots \ G_n]| \ B_2$$

dénote le comportement qui exécute B_1 et B_2 en parallèle. La synchronisation et la communication entre les opérandes B_1 et B_2 s'effectuent uniquement par rendez-vous sur les portes de l'ensemble $\{G_0, \ldots \ G_n, \delta\}$.

Lorsqu'un des opérandes veut effectuer une transition étiquetée par une porte G de $\{G_0, \ldots \ G_n, \delta\}$, il doit attendre que l'autre opérande puisse en faire autant. Lorsque le rendez-vous est possible, les deux opérandes effectuent simultanément une même transition *synchrone* étiquetée G ; puis ils reprennent chacun leur exécution.

En revanche si l'un des opérandes veut effectuer une transition étiquetée par une porte G qui n'appartient pas à $\{G_0, \ldots \ G_n, \delta\}$, il le fait indépendamment de l'autre opérande, de manière *asynchrone*.

Remarque B-2
En CCS, le rendez-vous entre deux portes complémentaires est facultatif ; on peut le rendre obligatoire en utilisant l'opérateur de restriction, mais on ne peut jamais l'interdire.

En LOTOS si deux portes identiques sont composées de manière synchrone, le rendez-vous est obligatoire ; si elles sont composées de manière asynchrone, le rendez-vous est interdit. ■

Exemple B-6
Pour composer en parallèle le distributeur de boissons (exemple B-5 (p. 20)) et un consommateur de thé (exemple B-1 (p. 17)) il faut les synchroniser sur les quatre interactions `MONEY`, `TEA`, `COFFEE` et `CHOCOLATE`. Comme le client n'effectue pas les interactions `COFFEE` et `CHOCOLATE`, le distributeur, qui doit se synchroniser avec lui, ne le peut pas non plus.

```
MONEY;
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         stop
   )
|[MONEY, TEA, COFFEE, CHOCOLATE]|
   MONEY;
      TEA;
         stop
```

■

Outre l'opérateur de synchronisation général "|[$G_0, \ldots\ G_n$]|" qui traduit la synchronisation sur les portes $G_0, \ldots\ G_n$ et "δ", LOTOS possède deux autres opérateurs de composition parallèle :

- le premier opérateur exprime la synchronisation sur aucune porte, sauf "δ" (*interleaving*). Sa syntaxe est :

  $$B_1 \texttt{ ||| } B_2$$

  Les deux comportements B_1 et B_2 sont exécutés de manière totalement indépendante (terminaison sur "δ" exceptée) : il ne se synchronisent ni ne communiquent l'un avec l'autre. En revanche ils sont capables d'interagir avec leur environnement commun : "B_1 ||| B_2" peut participer à un rendez-vous si et seulement si B_1 *ou* B_2 le peut

- le second opérateur exprime la synchronisation sur toutes les portes, y compris "δ" (*full synchronisation*). Sa syntaxe est :

  $$B_1 \texttt{ || } B_2$$

  Les deux comportements B_1 et B_2 sont exécutés en parallèle de manière entièrement synchrone : ils doivent se synchroniser sur toutes leurs interactions. Ils peuvent interagir avec leur environnement commun : "B_1 || B_2" peut participer à un rendez-vous si et seulement si B_1 *et* B_2 le peuvent

Exemple B-7
Si l'on veut modéliser le comportement simultané de quatre utilisateurs du distributeur de boissons décrit dans l'exemple B-6 (p. 21), il faut employer l'opérateur "|||". En effet ces consommateurs (thé : 1, café : 2, chocolat : 1) sont en concurrence pour l'accès à la ressource commune constituée par la machine. En revanche, comme la machine ne peut servir qu'un seul client à la fois, le groupe des quatre consommateurs doit être synchronisé avec le distributeur sur toutes les portes (`MONEY`, `TEA`, `COFFEE` et `CHOCOLATE`) ; on peut donc employer l'opérateur "||" :

```
MONEY;
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         stop
   )
||
   (
   MONEY;
      TEA;
         stop
   |||
   MONEY;
      COFFEE;
         stop
   |||
   MONEY;
      COFFEE;
         stop
   |||
   MONEY;
      CHOCOLATE;
         stop
   )
```

■

Le tableau suivant indique pour chaque opérateur de composition parallèle et pour chaque porte de $\Gamma \cup \{\delta\} \cup \{\mathbf{i}\}$ si deux comportements composés en parallèle par cet opérateur doivent, oui ou non, se synchroniser sur cette porte.

portes	"|||"	"|[$G_0, \ldots\ G_n$]|"	"||"
δ	oui	oui	oui
$G_0, \ldots\ G_n$	non	oui	oui
$\Gamma - \{G_0, \ldots\ G_n\}$	non	non	oui
$\mathbf{i}$	non	non	non

Lorsque les portes sont accompagnées d'offres, c'est-à-dire quand la composition parallèle a la forme suivante :

$$(G_1\ O_0^1 \ldots\ O_n^1\ \texttt{[}V_1^1\texttt{=}V_2^1\texttt{]}\ \texttt{;}\ B_1')\ op\ (G_2\ O_0^2 \ldots\ O_p^2\ \texttt{[}V_1^2\texttt{=}V_2^2\texttt{]}\ \texttt{;}\ B_2')$$

le rendez-vous n'a lieu que si les conditions suivantes sont vérifiées :

- les portes G_1 et G_2 sont égales et leur synchronisation est permise par l'opérateur *op*
- le nombre d'offres de part et d'autre est le même ($n = p$)
- les sortes des offres O_i^1 et O_i^2 sont deux à deux identiques
- les deux gardes sont vérifiées

Lorsqu'il y a confrontation entre une émission (offre "`!`") et une réception (offre "`?`") la valeur émise est affectée à la variable de réception (*value passing*).

Exemple B-8
C'est le cas lorsqu'on compose en parallèle un distributeur qui rend la monnaie (exemples B-2 (p. 18) et B-6 (p. 21)) et un buveur de thé qui fournit \$1.00 à la machine et reprend sa monnaie.

```
MONEY ?DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST];
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         CHANGE !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
            stop
   )
|[MONEY, TEA, COFFEE, CHOCOLATE, CHANGE]|
MONEY !1 !0;
   TEA;
      CHANGE ?DOLLARS:NAT ?CENTS:NAT;
         stop
```

■

LOTOS autorise également les confrontations entre deux émissions ("!" et "!") ou deux réceptions ("?" et "?"). Dans le premier cas (*value matching*), le rendez-vous n'a lieu que si les deux valeurs émises sont égales. Dans le second cas (*value generation*), les deux variables de réception reçoivent une valeur identique, choisie de manière non-déterministe. Le tableau suivant résume ces différentes possibilités :

offre n° 1	*offre n° 2*	*condition*	*action*
$!V_1$	$!V_2$	$V_1 = V_2$	—
$!V_1$	$?X_2:S_2$	$V_1 \in domain(S_2)$	$X_2 := V_1$
$?X_1:S_1$	$!V_2$	$V_2 \in domain(S_1)$	$X_1 := V_2$
$?X_1:S_1$	$?X_2:S_2$	$S_1 = S_2$	$X_1, X_2 := V(V \in domain(S_1))$

LOTOS permet le rendez-vous n-aire, c'est-à-dire la synchronisation, sur un même événement, de n comportements concurrents.

Exemple B-9
On peut imaginer un percolateur qui délivre simultanément deux tasses de café après avoir accepté successivement deux pièces de monnaie. Il est utilisé par deux consommateurs qui se synchronisent sur la porte `COFFEE` (puisqu'ils doivent recevoir simultanément leur boisson) et ne se synchronisent pas sur la porte `MONEY` (puisqu'ils paient à tour de rôle). On a ainsi un rendez-vous à trois sur la porte `COFFEE` entre le percolateur et les deux clients :

```
MONEY;
   MONEY;
      COFFEE;
         stop
|[MONEY, COFFEE]|
   (
   MONEY;
      COFFEE;
         stop
   |[COFFEE]|
   MONEY;
      COFFEE;
         stop
   )
```

■

Au cours d'un rendez-vous n-aire, n offres $O_1, \ldots O_n$ peuvent s'unifier si et seulement si l'intersection des ensembles de valeurs permis par les offres $O_1, \ldots O_n$ est non vide (s'il y a des gardes, il faut se restreindre aux valeurs pour lesquelles les conditions des gardes sont vérifiées). La valeur échangée

est choisie de manière non-déterministe dans cette intersection. En particulier ce mécanisme permet de spécifier la *diffusion*, c'est-à-dire l'émission d'un message par un comportement et sa réception par les autres comportements.

Chaque opérateur parallèle est commutatif et associatif (l'associativité n'est pas vérifiée pour deux opérateurs parallèles différents) ; l'opérande “**stop**” n'est élément absorbant à gauche (*resp.* à droite) que si l'autre opérande n'effectue pas de transition étiquetée “**i**”.

B.7 Opérateur “par”

De la même manière qu'il existe un opérateur “**choice**” permettant d'écrire le choix non-déterministe sous une forme concise, LOTOS comporte un opérateur “**par**” adapté à la composition parallèle.

On note “*op*” un des trois opérateurs parallèle :

$$\begin{array}{rcl} op & \equiv & \texttt{||} \\ & | & \texttt{|||} \\ & | & \texttt{|[}\widehat{G}\texttt{]|} \end{array}$$

Soit B_0 un comportement qui contient des occurrences d'utilisation d'une porte G. Soient G_1, ... G_n des portes et soient B_1, ... B_n les comportements définis de la manière suivante : B_i est obtenu à partir de B en remplaçant G par G_i. Pour exprimer le comportement :

$$B_1 \ op \ \ldots \ B_n$$

LOTOS permet d'utiliser une notation abrégée :

$$\textbf{par } G \textbf{ in } [G_1, \ldots \ G_n] \ op \ B_0$$

La porte G sert d'indice à cette itération. Il n'est pas nécessaire que les portes G_1, ... G_n soient deux à deux distinctes.

Exemple B-10
On peut écrire l'exemple B-7 (p. 22) de manière plus concise :

```
MONEY;
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         stop
   )
||
   (
   par DRINK in [TEA, COFFEE, COFFEE, CHOCOLATE] |||
      MONEY;
         DRINK;
            stop
   )
```

■

Comme pour l'opérateur “**choice**”, il existe une forme plus générale permettant d'itérer sur plusieurs portes :

$$\textbf{par } \widehat{G_0} \textbf{ in } [\widehat{G'_0}], \ldots \ \widehat{G_n} \textbf{ in } [\widehat{G'_n}] \ op \ B_0$$

Les portes définies dans les listes $\widehat{G_0}$, ... $\widehat{G_n}$ ne sont visibles que dans B_0.

B.8 Opérateur "hide"

LOTOS possède un opérateur qui permet de cacher certaines portes d'un comportement. Si $G_0, \ldots G_n$ sont des portes et B_0 un comportement, la construction suivante :

$$\textbf{hide } G_0, \ldots G_n \textbf{ in } B_0$$

dénote le comportement B_0 dont les portes $G_0, \ldots G_n$ sont renommées en "**i**". Les portes $G_0, \ldots G_n$ ne sont visibles que dans B_0. Elles deviennent inaptes à la synchronisation pour l'environnement de B_0 avec lequel elles n'interfèrent plus. Vu de l'extérieur ces interactions sont invisibles (puisqu'étiquetées "**i**") et ont lieu spontanément sans aucune participation de l'environnement de B_0 : le rendez-vous sur une porte cachée n'est jamais bloquant.

Exemple B-11
Bien souvent un comportement est décrit comme la mise en parallèle de plusieurs sous-comportements qui se synchronisent sur un ensemble de portes qu'il convient de dissimuler vis-à-vis de l'environnement. Dans cet esprit, on peut décomposer le distributeur décrit dans l'exemple B-8 (p. 23) en deux sous-systèmes :

- le premier reçoit une somme d'argent, s'assure que le montant est suffisant, calcule la monnaie à rendre et envoie une autorisation à l'autre sous-système via une porte `GRANT`
- le second, lorsque l'autorisation est accordée, délivre une boisson (thé, café ou chocolat, au choix du client) et rend la monnaie (la somme qu'il faut restituer lui a été communiquée via la porte `GRANT`)

On cache la porte `GRANT` au moyen de l'opérateur "**hide**" car il s'agit d'un détail d'implémentation qui n'est pas pertinent pour un observateur extérieur.

Le distributeur est composé en parallèle avec un consommateur de thé qui fournit $1.00 pour payer. Noter que l'opérateur "`||`" impose la synchronisation sur les portes `MONEY`, `TEA`, `COFFEE`, `CHOCOLATE` et `CHANGE` mais pas `GRANT`, qui est cachée.

```
hide GRANT in
   (
   MONEY ?DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST];
      GRANT !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
         stop
   |[GRANT]|
   GRANT ?DOLLARS:NAT ?CENTS:NAT;
      (
      choice DRINK in [TEA, COFFEE, CHOCOLATE] []
         DRINK;
            CHANGE !DOLLARS !CENTS;
               stop
      )
   )
||
   MONEY !1 !0;
      TEA;
         CHANGE ?DOLLARS:NAT ?CENTS:NAT;
            stop
```

■

Le rendez-vous "unaire" est correct et non bloquant ; par exemple :

$$\textbf{hide } G \textbf{ in } G \textbf{ ; stop}$$

est équivalent, vu de l'extérieur, à :

i ; stop

Remarque B-3
On peut comparer les solutions retenues pour CCS et LOTOS : CCS ne possède pas d'opérateur d'abstraction ("**hide**" en LOTOS) et, inversement, LOTOS ne possède pas d'opérateur de restriction ("\" en CCS).

En CCS l'abstraction est couplée avec l'opérateur parallèle : après synchronisation les portes complémentaires G et $\overline{G}$ sont cachées ; ce choix interdit le rendez-vous n-aire mais permet les rendez-vous à 2 parmi n.

En LOTOS la restriction est couplée avec l'opérateur parallèle : si cet opérateur indique qu'une porte G doit être synchronisée, alors tout rendez-vous sur G est bloquant. Cette méthode autorise le rendez-vous n-aire ; en revanche le rendez-vous à 2 parmi n est plus difficile à obtenir, mais plusieurs solutions existent néanmoins. ■

B.9 Opérateur "->"

LOTOS permet de conditionner tout comportement par une garde qui est, soit une expression booléenne, soit une équation simple. Si V_0, V_1 et V_2 sont des expressions de valeur et B_0 une expression de comportement, les deux constructions :

$$[V_0] \texttt{ -> } B_0$$

et :

$$[V_1\texttt{=}V_2] \texttt{ -> } B_0$$

dénotent un comportement qui est égal à B_0 si la condition de garde est vraie et à "**stop**" si elle est fausse.

LOTOS ne possède pas de clause "**if then else**" : il faut utiliser des *commandes gardées*, obtenues en combinant l'opérateur "`->`" avec l'opérateur "`[]`", comme le montre l'exemple suivant.

Exemple B-12
Le distributeur décrit dans l'exemple B-8 (p. 23) peut être modifié afin qu'il accepte tout paiement quel que soit son montant mais, s'il est insuffisant, il restitue la somme sans délivrer de boisson :

```
MONEY ?DOLLARS:NAT ?CENTS:NAT;
   (
   [TOTAL (DOLLARS, CENTS) lt COST] ->
      CHANGE !DOLLARS !CENTS;
         stop
   []
   [TOTAL (DOLLARS, CENTS) ge COST] ->
      (
      choice DRINK in [TEA, COFFEE, CHOCOLATE] []
         DRINK;
            CHANGE !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
               stop
      )
   )
```

■

Si, dans une commande gardée de la forme :

$$\begin{array}{ll} & \texttt{[}V_0\texttt{]} \texttt{ -> } B_0 \\ \texttt{[]} & \texttt{[}V_1\texttt{]} \texttt{ -> } B_1 \\ & \ldots \\ \texttt{[]} & \texttt{[}V_n\texttt{]} \texttt{ -> } B_n \end{array}$$

les conditions $V_0, \ldots V_n$ ne sont pas mutuellement exclusives, le non-déterministe s'applique.

Remarque B-4
Le comportement :

$$G_1\ O_0^1 \ldots\ O_n^1\ \texttt{[}V_1^1\texttt{=}V_2^1\texttt{]}\ \texttt{;}\ \textbf{stop}$$

n'est pas équivalent à :

$$G_1\ O_0^1 \ldots\ O_n^1\ \texttt{;}\ \texttt{[}V_1^1\texttt{=}V_2^1\texttt{]}\ \texttt{->}\ \textbf{stop}$$

car dans le second cas le rendez-vous sur la porte G a toujours lieu, même si la garde est fausse. ■

Remarque B-5
L'opérateur "`[]`" de LOTOS possède de "bonnes propriétés" qui limitent le risque de blocage. En particulier la garde est distributive sur le choix non-déterministe. Autrement dit, le comportement :

$$\texttt{(}\texttt{[}V\texttt{]}\texttt{ -> }B_1\texttt{)}\ \texttt{[]}\ \texttt{(}\texttt{[}V\texttt{]}\texttt{ -> }B_2\texttt{)}$$

est équivalent à :

$$\texttt{[}V\texttt{]}\texttt{ -> (}B_1\ \texttt{[]}\ B_2\texttt{)}$$

En CSP et dans les autres langages où la garde n'est pas distributive sur la composition non-déterministe, les comportements de ce genre sont généralement incorrects. En effet, si l'on écrit en CSP [**?**] un comportement de la forme :

$$\begin{array}{l} \textbf{true}\texttt{ -> }G_1\ \texttt{?}X_1\ \texttt{;}\ B_1 \\ \texttt{[]} \\ \textbf{true}\texttt{ -> }G_2\ \texttt{?}X_2\ \texttt{;}\ B_2 \end{array}$$

on obtient un choix non-déterministe qui n'est pas contrôlable par l'environnement. ■

B.10 Opérateur "let"

Lorsqu'une expression de valeur doit être utilisée plusieurs fois, il est possible de lui donner un nom grâce à une définition de variable. Si V est une expression de valeur de sorte S, la construction suivante :

$$\textbf{let}\ X\texttt{:}S\texttt{=}V\ \textbf{in}\ B_0$$

dénote le comportement obtenu à partir de B_0 dans lequel la variable X possède la valeur V.

Exemple B-13
Le distributeur décrit dans l'exemple B-12 (p. 27) peut être simplifié en utilisant une variable booléenne `OK` pour factoriser les expressions figurant dans les gardes :

```
MONEY ?DOLLARS:NAT ?CENTS:NAT;
   (
   let OK:BOOL=(TOTAL (DOLLARS, CENTS) ge COST) in
      (
      [not (OK)] ->
         CHANGE !DOLLARS !CENTS;
            stop
      []
      [OK] ->
         (
         choice DRINK in [TEA, COFFEE, CHOCOLATE] []
            DRINK;
               CHANGE !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS);
                 stop
         )
      )
   )
```

■

L'opérateur “**let**” possède une forme plus générale permettant de définir simultanément plusieurs variables :

$$\textbf{let } \widehat{X_0}:S_0{=}V_0, \ldots \ \widehat{X_n}:S_n{=}V_n \textbf{ in } B_0$$

Les variables définies dans les listes $\widehat{X_0}$, … $\widehat{X_n}$ ne sont visibles que dans B_0. Chaque variable de la liste $\widehat{X_i}$ est de sorte S_i et a pour valeur V_i.

B.11 Opérateur “choice” sur les valeurs

Soit B_0 un comportement qui contient des occurrences d'utilisation d'une variable X de sorte S. Soit B_V le comportement obtenu à partir de B_0 en donnant à X la valeur V. Pour exprimer le comportement :

$$B_{V_1} \ \texttt{[]} \ \ldots \ B_{V_n} \ \texttt{[]} \ \ldots$$

où $V_1, \ldots \ V_n, \ldots$ désignent les valeurs de $domain(S)$, LOTOS permet d'utiliser une notation abrégée :

$$\textbf{choice } X:S \ \texttt{[]} \ B_0$$

La variable X sert d'indice à cette itération sur les valeurs du domaine de S.

Exemple B-14
Il est possible de modifier la façon dont le distributeur présenté dans l'exemple B-8 (p. 23) rend la monnaie afin qu'il ne restitue pas systématiquement le nombre maximal `D_MAX` de dollars mais un nombre `D` choisi entre `0` et `D_MAX` de manière non-déterministe (le distributeur complète avec autant de cents qu'il le faut) :

```
MONEY ?DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST];
   (
   choice DRINK in [TEA, COFFEE, CHOCOLATE] []
      DRINK;
         (
         let D_MAX:NAT=CHG_DOLLARS (DOLLARS, CENTS) in
            choice D:NAT []
               [D le D_MAX] ->
                  (
                  let C:NAT=TOTAL (D_MAX - D, CHG_CENTS (DOLLARS, CENTS)) in
                     CHANGE !D !C;
                        stop
                  )
         )
   )
```

■

L'opérateur "**choice**" possède une forme plus générale permettant de définir plusieurs variables :

$$\textbf{choice}\ \widehat{X_0}{:}S_0, \ldots\ \widehat{X_n}{:}S_n\ \texttt{[]}\ B_0$$

Les variables définies dans les listes $\widehat{X_0}$, ... $\widehat{X_n}$ ne sont visibles que dans B_0. Chaque variable de la liste $\widehat{X_i}$ est de sorte S_i (les variables d'une même liste ne reçoivent pas forcément la même valeur).

Remarque B-6
Dans les offres de rendez-vous, une émission (offre "`!`") peut être vue comme un cas particulier de réception (offre "`?`") où le domaine des valeurs acceptées est réduit à un singleton. Réciproquement toute réception peut être exprimée comme une émission en utilisant l'opérateur "**choice**" sur les valeurs. C'est ainsi que le comportement :

$$G\ \texttt{?}X{:}S\ \texttt{[}V\texttt{]}\ \texttt{;}\ B_0$$

est équivalent à :

$$\textbf{choice}\ X{:}S\ \texttt{[]}\ \texttt{([}V\texttt{]}\ \texttt{->}\ G\ \texttt{!}X\ \texttt{;}\ B_0\texttt{)}$$

■

Remarque B-7
Il existe deux formes de l'opérateur "**choice**" : une pour les portes (§ B.5, p. 20) l'autre pour les valeurs (§ B.11, p. 29). En revanche l'opérateur "**par**" n'existe que pour les portes (§ B.7, p. 25). ■

B.12 Opérateur "exit"

L'opérateur "**stop**" permet de spécifier explicitement l'arrêt d'un comportement. Mais "**stop**" peut aussi apparaître de manière implicite, lorsqu'un comportement se bloque. Pour distinguer ces deux formes de terminaison, normale et anormale, on introduit un nouvel opérateur. La construction suivante :

exit

dénote un comportement qui se termine normalement. La terminaison avec succès s'exprime par le franchissement d'une transition "δ". De fait "**exit**" est équivalent à un rendez-vous sur la porte "δ" :

δ **; stop**

Un comportement peut, lorsqu'il se termine par “**exit**”, transmettre des *résultats*. Cette possibilité correspond à l'ajout d'une liste d'offres au rendez-vous sur la porte “δ”, mais la syntaxe est différente :

$$\textbf{exit}\ (R_0, \ldots\ R_n)$$

où les résultats R_0, ... R_n sont définis comme suit :

$$\begin{array}{rcl} R & \equiv & V \\ & | & \textbf{any}\ S \end{array}$$

Un résultat dénote donc, soit une valeur V déterminée, soit une valeur choisie de façon non-déterministe dans le domaine d'une sorte S.

La définition de l'opérateur de composition parallèle (§ B.6, p. 21) et de la synchronisation sur la porte “δ” implique que la composition parallèle de n comportements B_1, ... B_n ne se termine par “**exit**” que si B_1, ... B_n se terminent aussi par “**exit**”, de manière synchrone (*join*), en proposant des offres compatibles.

Certaines règles interdisent les constructions susceptibles de conduire à des blocages sur la porte “δ”. Savoir si un comportement se termine ou non étant un problème indécidable dans le cas général, LOTOS se contente d'imposer des contraintes “de bon sens”, qu'il est possible de vérifier statiquement et qui protègent l'utilisateur contre certaines erreurs. Chaque comportement possède une *fonctionnalité* qui spécifie si le comportement se termine et précise, dans ce cas, les sortes des valeurs qu'il renvoie par l'opérateur “**exit**”. Il faut respecter certaines règles quand on compose les comportements ; c'est ainsi qu'il est interdit d'écrire :

$$\textbf{exit}\ (V_1, \ldots\ V_n)\ \texttt{|||}\ \textbf{exit}\ (V'_1, \ldots\ V'_{n'})$$

lorsque n et n' ne sont pas égaux.

B.13 Opérateur “>>”

L'opérateur de préfixage “**;**” est asymétrique : son opérande gauche est une porte (éventuellement accompagnée d'offres) alors que son opérande droit est un comportement. LOTOS possède un autre opérateur de composition séquentielle dont les deux opérandes sont des comportements. Si B_1 et B_2 sont deux comportements, la construction suivante :

$$B_1\ \texttt{>>}\ B_2$$

dénote le comportement qui exécute séquentiellement B_1 puis B_2.

Intuitivement, la composition séquentielle est modélisée en LOTOS comme un cas particulier de composition parallèle. Les comportements B_1 et B_2 sont exécutés en parallèle mais B_2 ne peut pas commencer avant que B_1 ne se soit terminé. L'attente de B_2 s'obtient par un rendez-vous sur la porte “δ”, rendez-vous qui devient possible dès que B_1 exécute “**exit**”. Si B_1 boucle indéfiniment ou se bloque sans atteindre “**exit**” B_2 ne sera jamais exécuté.

L'opérateur “**>>**” est souvent appelé *enabling operator* ou *enable* puisque la terminaison avec succès de B_1 autorise l'exécution de B_2.

L'opérateur “**>>**” possède une forme plus générale permettant au processus qui commence de récupérer les résultats renvoyés par le processus qui se termine. Si B_1 et B_2 sont deux comportements, la

construction suivante :

$$B_1 \texttt{ >> } \textbf{accept } \widehat{X_0}:S_0, \ldots \ \widehat{X_n}:S_n \textbf{ in } B_2$$

dénote le comportement formé par la composition séquentielle de B_1 de B_2 ; lorsque B_1 exécute une instruction "**exit**", les résultats qu'il renvoie sont affectés aux variables $\widehat{X_0}$, ... $\widehat{X_n}$ respectivement. Les variables définies dans les listes $\widehat{X_0}$, ... $\widehat{X_n}$ ne sont visibles que dans B_2. Chaque variable de la liste $\widehat{X_i}$ est de sorte S_i (les contraintes portant sur la fonctionnalité des comportements imposent que les résultats renvoyés par B_1 correspondent, par leur nombre et par leurs sortes, aux variables déclarées après le mot "**accept**").

Exemple B-15
On peut réécrire le distributeur présenté dans l'exemple B-13 (p. 28) en factorisant la restitution de monnaie (interaction `CHANGE`) grâce à l'opérateur de composition séquentielle :

```
MONEY ?DOLLARS:NAT ?CENTS:NAT;
   (
   let OK:BOOL=(TOTAL (DOLLARS, CENTS) ge COST) in
      (
      [not (OK)] ->
         exit (DOLLARS, CENTS)
      []
      [OK] ->
         (
         choice DRINK in [TEA, COFFEE, CHOCOLATE] []
            DRINK;
               exit (CHG_DOLLARS (DOLLARS, CENTS), CHG_CENTS (DOLLARS, CENTS))
         )
      )
   )
>> accept DOLLARS, CENTS:NAT in
   CHANGE !DOLLARS !CENTS;
      stop
```

■

L'opérateur "`>>`", avec ou sans "**accept**", est associatif.

B.14 Opérateur "[>"

LOTOS dispose d'un opérateur permettant de spécifier l'interruption d'un comportement par un autre. Si B_1 et B_2 sont deux comportements, la construction suivante :

$$B_1 \texttt{ [> } B_2$$

dénote le comportement qui exécute B_1 mais qui peut abandonner à tout instant l'exécution de B_1 pour commencer celle de B_2.

Intuitivement, il s'agit d'un mécanisme d'interruption avec terminaison : B_1 joue le rôle d'un traitement normal et B_2 celui d'un traitement d'exception. Initialement, B_1 est exécuté seul mais, tant qu'il n'a pas effectué de transition "δ", son exécution peut être interrompue au profit de celle de B_2. Si B_1 se bloque avant d'atteindre une instruction "**exit**" alors B_2 est inévitablement exécuté. Dans le cas contraire B_2 peut très bien ne jamais être exécuté.

L'opérateur "`[>`" est souvent appelé *disabling operator* ou *disable* puisque la terminaison avec succès de B_1 interdit l'exécution de B_2.

Exemple B-16
L'opérateur "[>" peut être utilisé pour décrire le comportement d'un consommateur de thé lorsque le distributeur comporte un bouton d'annulation (interaction CANCEL). Le comportement normal du consommateur est celui décrit dans l'exemple B-8 (p. 23) mais, à chaque instant, il peut s'interrompre et presser sur le bouton d'annulation pour tenter de se faire rembourser par le distributeur (qui n'est pas obligé d'accepter).

```
MONEY !1 !0;
   TEA;
      CHANGE ?DOLLARS:NAT ?CENTS:NAT;
         exit
[>
CANCEL;
   CHANGE ?DOLLARS:NAT ?CENTS:NAT;
      exit
```

■

L'opérateur "[>" est associatif.

B.15 Processus et instanciation

Dans les langages algorithmiques, il est possible de donner un nom à un bloc d'instructions, en définissant une *procédure* qui peut éventuellement être paramétrée. De manière analogue, LOTOS permet de nommer un comportement au moyen d'une définition de *processus* (*process*). Un processus est un objet qui dénote un comportement ; il peut être paramétré par une liste de *portes formelles* et/ou une liste de *variables formelles*.

Remarque B-8
Ce mécanisme est limité au 1$^{\text{er}}$ ordre : il n'existe pas d'objet qui dénote un processus. On n'a donc pas de variables ou de paramètres formels "de type processus". ■

Remarque B-9
En LOTOS le mot "**process**" n'a pas le même sens que dans d'autres langages ; il ne dénote pas forcément une activité concurrente. La création des tâches parallèles est dévolue à l'opérateur de composition parallèle et non à l'instanciation. ■

La construction suivante :

process P [[$G_0, \ldots\ G_m$]] [($\widehat{X_0}$:$S_0, \ldots\ \widehat{X_n}$:$S_n$)] : *func* :=
  B
  [**where** $block_0, \ldots\ block_p$]
endproc

définit un processus P (éventuellement paramétré par les portes formelles $G_1, \ldots\ G_m$ et les listes de variables formelles $\widehat{X1}, \ldots\ \widehat{X_n}$ de sortes respectives $S_1, \ldots\ S_n$) dont le *corps* est le comportement B. Le non-terminal *func* spécifie la fonctionnalité de B afin de permettre une vérification statique des contraintes de fonctionnalité ; sa définition syntaxique est :

func ≡ **noexit**
  | **exit** [($S_0, \ldots\ S_n$)]

Le premier cas indique que B ne s'achève jamais par "**exit**" (ce qui signifie que B se bloque ou boucle indéfiniment) ; le second cas exprime que B se termine en renvoyant des résultats de sortes respectives

$S_0, \ldots S_n$. Chaque non-terminal $block_i$ dénote une définition de processus ou de type ; dans les deux cas il s'agit d'une définition locale dont la visibilité est limitée à la définition du processus P.

L'instanciation d'un processus s'effectue en substituant des paramètres effectifs aux paramètres formels. Si P est un processus, $G_0, \ldots G_m$ des portes et $V_0, \ldots V_n$ des expressions de valeur, la construction suivante :

$$P\ [[G_0, \ldots\ G_m]]\ [(V_0, \ldots\ V_n)]$$

dénote le corps de P dans lequel les portes formelles sont renommées par $G_0, \ldots G_m$ et les variables formelles sont instanciées avec les valeurs $V_0, \ldots V_n$.

L'emploi de la récursion est autorisé ; en LOTOS la récursion est d'ailleurs le seul moyen pour créer des comportements cycliques.

Exemple B-17
L'exemple suivant décrit un distributeur relativement élaboré. Le consommateur peut effectuer autant de paiements qu'il le souhaite (plusieurs interactions `MONEY` successives) jusqu'à ce que le prix d'une boisson `COST` soit atteint ; il peut même dépasser ce montant.

Dès que la somme versée est suffisante, le consommateur peut choisir une boisson et récupérer sa monnaie, comme dans l'exemple B-8 (p. 23). L'utilisateur a aussi la possibilité de récupérer l'argent versé en appuyant sur le bouton `CANCEL` comme dans l'exemple B-16 (p. 33).

Le fonctionnement du distributeur est cyclique : il retourne dans son état initial et peut donc servir successivement plusieurs clients.

```
   SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
where
process SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (D, C:NAT) : noexit :=
   MONEY ?DOLLARS:NAT ?CENTS:NAT;
      (
      let D0:NAT=(D + DOLLARS), C0:NAT=(C + CENTS) in
         (
         [TOTAL (D0, C0) ge COST] ->
            (
            choice DRINK in [TEA, COFFEE, CHOCOLATE] []
               DRINK;
                  CHANGE !CHG_DOLLARS (D0, C0) !CHG_CENTS (D0, C0);
                     SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
            )
         []
         CANCEL;
            CHANGE !D0 !C0;
               SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
         []
         SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (D0, C0)
         )
      )
endproc
```

■

Exemple B-18
Pour décrire un distributeur qui délivre successivement trois sortes de boissons, on peut employer une permutation circulaire des paramètres portes :

```
   SELL [MONEY, TEA, COFFEE, CHOCOLATE]
where
process SELL [MONEY, DRINK_1, DRINK_2, DRINK_3] : noexit :=
   MONEY;
      DRINK_1;
         SELL [MONEY, DRINK_2, DRINK_3, DRINK_1]
endproc
```

■

Il est possible de spécifier en LOTOS la création et la destruction dynamique de processus concurrents ; on emploie pour cela la récursion en partie gauche ou droite d'un opérateur de composition parallèle.

B.16 Spécification LOTOS

Un programme (ou *spécification*) LOTOS est assimilable à la définition d'un processus qui englobe toutes les autres définitions de types et de processus. Par rapport à une définition de processus la syntaxe varie légèrement :

specification λ [[$G_0, \ldots\ G_m$]] [($\widehat{X_0}$:$S_0, \ldots\ \widehat{X_n}$:$S_n$)] : *func*
  type$_1, \ldots$ *type*$_p$
  behaviour B
  [**where** *block*$_0, \ldots$ *block*$_q$]
endspec

Chaque non-terminal *type*$_i$ dénote une définition de type dont la visibilité s'étend à toute la spécification ; en particulier les sortes $S_0, \ldots\ S_n$ doivent être définies dans ces types. Le comportement B est le corps de la spécification. Le non-terminal *func* spécifie la fonctionnalité de B. Chaque non-terminal *block*$_i$ dénote une définition de type ou de processus.

L'identificateur λ joue le rôle d'un commentaire ; ce n'est pas un identificateur de processus, il ne peut donc pas être utilisé dans une instanciation.

B.17 Styles de programmation

Le langage LOTOS autorise plusieurs styles de spécification :

programmation logique et algébrique : on dispose pour cela, des types abstraits et des équations algébriques

programmation fonctionnelle : on peut modéliser les fonctions à décrire par des processus possédant des paramètres variables et retournant leurs résultats grâce à l'opérateur "**exit**", en se restreignant à l'emploi des opérateurs "**let**", "`->`", "`[]`", "`>>`" et l'instanciation, sans utiliser aucune porte[3] ni aucun rendez-vous

Exemple B-19
Le processus LOTOS suivant calcule le quotient et le reste de la division euclidienne de deux entiers :

[3] hormis la porte implicite δ

```
process DIV_MOD (X, Y:NAT) : exit (NAT, NAT) :=
   [X lt Y] ->
      exit (0, X)
   []
   [X ge Y] ->
      (
      DIV_MOD (X - Y, Y)
      >> accept Q, R:NAT in
         exit (Q + 1, R)
      )
endproc
```

■

programmation parallèle : il s'agit du style de programmation dual du précédent. En programmation fonctionnelle le contrôle est simplifié à l'extrême ; l'approche inverse consiste à n'avoir aucune structure de données, à se limiter à la synchronisation pure et à représenter tous les objets (piles, files, ...) comme des réseaux de processus communicants dont la structure évolue de manière dynamique

programmation objet : nombre de langages effectuent une distinction entre les concepts d'encapsulation et de parallélisme ("**module/process**" en MODULA-2 [**?**], "**package/task**" en ADA) alors que ces deux notions possèdent des caractéristiques voisines : un module change d'état quand les procédures qu'il exporte sont appelées ; un processus change d'état lorsque les rendez-vous qu'il propose sont acceptés.

Un objet peut être modélisé par un comportement LOTOS ; les primitives de manipulation de l'objet correspondent aux rendez-vous acceptés par le comportement. On peut ainsi imposer des restrictions sur l'ordre dans lequel les primitives de l'objet sont appelées et refuser un rendez-vous si une contrainte n'est pas respectée. Par le biais des interactions cachées on peut effectuer automatiquement certaines actions d'initialisation, de terminaison, ...

Exemple B-20
L'exemple suivant contient la spécification en LOTOS d'une pile de nombres naturels. La structure de données qui implémente l'état de la pile est décrite par le type abstrait `STACK_DATA`. Le processus `STACK_CONTROL` constitue une interface qui accepte trois sortes de messages émis par l'utilisateur de la pile :

- `RESET` : initialiser ou ré-initialiser la pile
- `PUSH !`V : empiler la valeur V au sommet de la pile
- `POP ?`X`:NAT` : désempiler la valeur se trouvant au sommet et l'affecter à la variable X

```
process STACK [RESET, PUSH, POP] : noexit :=
   RESET;
      STACK_CONTROL [RESET, PUSH, POP] (VOID)
where
   process STACK_CONTROL [RESET, PUSH, POP] (S:STACK) : noexit :=
      RESET;
         STACK_CONTROL [RESET, PUSH, POP] (VOID)
      []
      [not (FULL (S))] ->
         PUSH ?X:NAT;
            STACK_CONTROL [RESET, PUSH, POP] (INSERT (X, S))
      []
      [not (EMPTY (S))] ->
         POP !(TOP (S));
            STACK_CONTROL [RESET, PUSH, POP] (REMOVE (S))
   endproc

   type STACK_DATA is NATURALNUMBER, BOOLEAN
      sorts STACK
      opns VOID : -> STACK
           TOP : STACK -> NAT
           INSERT : NAT, STACK -> STACK
           REMOVE : STACK -> STACK
           FULL : STACK -> BOOL
           EMPTY : STACK -> BOOL
   endtype
endproc
```

■