Introduction à E-LOTOS

Guy Leduc, Alan Jeffrey, Mihaela Sighireanu

Chapter 6 : Introduction à E-LOTOS

G. Leduc, A. Jeffrey et M. Sighireanu[1]

Résumé : Cette annexe a pour but de présenter les éléments principaux du langage ISO E-LOTOS, qui est une révision du langage ISO LOTOS. Les améliorations apportées par E-LOTOS sont entre autres : la modélisation du temps, un système de modules, des types de données fonctionnels, la gestion d'exceptions, la présence de constructions impératives, et quelques nouveaux opérateurs ou extensions d'opérateurs. E-LOTOS est présenté en deux parties : le langage de base et le langage des modules.

Mots-clés : E-LOTOS, tutoriel

1. Introduction

Le langage formel ISO LOTOS [BOL 87, ISO 89] est composé d'une algèbre de processus (basée sur CCS [MIL 89] et CSP [HOA 85]) pour décrire les comportements, et d'un langage algébrique (ACT ONE [EHR 85]) pour décrire les types abstraits de donnée. Ce langage est mathématiquement bien défini et expressif : il permet la description du parallélisme, du non déterminisme, et des communications synchrones et asynchrones. Les spécifications peuvent être écrites à différents niveaux d'abstraction et dans différents styles. De bons outils existent pour vérifier les spécifications et générer du code. Malgré cela, LOTOS a été révisé par l'ISO [ISO 00] afin de tenir compte des remarques des utilisateurs qui ont critiqué quelques aspects techniques du langage, ainsi que sa convivialité en pratique.

[1]Guy Leduc : Université de Liège, Institut Montefiore, B28, B-4000 Liège, Belgique, courriel : leduc@montefiore.ulg.ac.be. Alan Jeffrey : DePaul University, 243 S. Wabash Ave, Chicago 60604, USA, courriel : ajeffrey@cs.depaul.edu. Mihaela Sighireanu : Université Denis Diderot (Paris 7) - LIAFA, 2, place Jussieu F-75251 Paris Cedex 5, courriel : Mihaela.Sighireanu@liafa.jussieu.fr

Une connaissance minimale de LOTOS est présupposée. Toutefois, nous rappelons la syntaxe du LOTOS de base (i.e. LOTOS sans types de données) et en donnons une description sommaire. Un *processus* LOTOS (Π) est défini par la liste des *portes de communication* (G) avec son environnement et par son *comportement* (B). Les comportements sont construits par l'algèbre de processus LOTOS selon la syntaxe suivante :

$$\begin{array}{rcl} B & ::= & \mathbf{stop} \mid \mathbf{exit} \mid \Pi[G^*] \mid G;B \mid \mathbf{i};B \mid B \ \texttt{[]} \ B \mid B \ \texttt{|[}G^*\texttt{]|} \ B \mid \\ & & \mathbf{hide}\ G^*\ \mathbf{in}\ B \mid B \gg B \mid B \ \texttt{[>}\ B \end{array}$$

La sémantique informelle peut être définie comme suit :

- Blocage : **stop** est un comportement inactif.
- Termination : **exit** est un comportement qui se termine avec succès par l'exécution d'une action sur la porte de terminaison δ avant de se bloquer.
- L'instanciation de processus : $\Pi[G^*]$ instancie la définition de ce processus avec les paramètres portes G^*.
- Le préfixage par une action : $G;B$ est un comportement qui exécute l'action de communication par G, puis se comporte comme B.
- Le préfixage par action interne : $\mathbf{i};B$ est un comportement qui exécute l'action interne **i**, puis se comporte comme B.
- Le choix externe : B_1 `[]` B_2 est un processus qui se comporte comme B_1 ou comme B_2 selon l'environnement.
- Parallélisme : B_1 `|[`G^*`]|` B_2 est la composition parallèle de B_1 et B_2 avec synchronisation sur les portes dans l'ensemble G^*.
- Abstraction : **hide** G^* **in** B cache toutes les actions effectuant des communications sur les portes de l'ensemble G^* dans le comportement B, c'est-à-dire que ces actions sont renommées par l'action interne **i**.
- Activation : $B_1 \gg B_2$ est la composition séquentielle de B_1 et B_2, c'est-à-dire que B_2 peut démarrer quand B_1 a terminé avec succès.
- Désactivation : B_1 `[>` B_2 permet à B_2 de désactiver B_1 pour autant que B_1 ne se soit pas terminé avec succès.

Les deux améliorations substantielles de E-LOTOS par rapport à LOTOS portent sur les types de données et sur la modélisation du temps. En effet, LOTOS ne permet pas de décrire des systèmes temps réel. De plus, les types de données algébriques ne sont pas très conviviaux, ni modulaires ; ils peuvent

conduire à des spécifications équationnelles indécidables et ils ne permettent pas de définir des opérations partielles.

Par exemple, un simple routeur de paquets composés d'un champ de donnée et d'une adresse était typiquement défini comme suit en LOTOS :

```
process Router [inp, left, right] : noexit :=
  inp?p:packet;
  (     [getdest(p) = L] -> left!getdata(p);Router [inp, left, right]
    [] [getdest(p) = R] -> right!getdata(p);Router [inp, left, right]
  )
endproc
```

Cette définition n'est pas très lisible pour un non expert en LOTOS. Ceci est dû à l'usage conjoint de l'opérateur de choix et de prédicats de sélection pour spécifier un simple opérateur **if-then-else**. Le pire est toutefois ailleurs. La description d'une structure de donnée aussi simple que le paquet ('packet') conduirait à la spécification suivante :

```
type Packet is Data
  sorts
    packet, dest
  opns
    mkpacket : dest,data -> packet
    getdest : packet -> dest
    getdata : packet -> data
    L :  -> dest
    R :  -> dest
  eqns forall p:packet, de:dest, da:data
    ofsort packet mkpacket (getdest (p),getdata (p)) = p
    ofsort dest getdest (mkpacket (de, da)) = de
    ofsort data getdata (mkpacket (de, da)) = da
  endeqns
endtype
```

En E-LOTOS, le même système pourrait être décrit comme suit :

```
type dest is L | R endtype
type packet is (de=>dest,da=>data) endtype
process Router [inp:packet,left:data,right:data] is
  var p:packet
  in
    inp ?p;
    case p.de is
          L -> left !p.da
       | R -> right !p.da
    endcase;
    Router [inp, left, right]
  endvar
endproc
```

On remarquera que :

- Les portes du processus 'Router' sont explicitement typées.
- Les champs du paquets sont accessibles par des fonctions prédéfinies de syntaxe simple, plutôt que par des opérations de sélection à créer explicitement.
- La portée de la variable 'p' est rendue explicite par une déclaration locale de variable (**var**).
- L'instruction **case** est rendue explicite, alors qu'elle n'était qu'implicite par l'usage conjoint des prédicats de sélection et de l'opérateur de choix.
- L'appel récursif a été déplacé en dehors de l'opérateur **case**, évitant ainsi de le dédoubler.
- Les définitions du type 'dest' comme une union, et du type 'packet' comme un enregistrement sont explicites, et nettement plus courtes.

Le langage E-LOTOS est un langage à deux niveaux. Le niveau supérieur est le langage des modules, qui sera décrit dans la section 3. Le niveau inférieur est le langage de base, qui se décompose à son tour dans un langage pour la spécification des données et un langage pour la spécification des comportements. Ces deux deux parties du langage de base partagent beaucoup de leur constructions, car les expressions de données constituent une sous-classe de comportements. Le langage de base sera décrit dans la section 2. Comme LOTOS, E-LOTOS est mathématiquement bien défini. Pour le langage de base sont définies les sémantiques statique et dynamique des expressions de données et de comportements. Pour le langage de modules est définie la sémantique

statique des modules et des interfaces, ainsi que la mise à plat des modules. Nous présenterons plus de details sur la sémantique à la fin de chaque section.

Le langage E-LOTOS décrit dans la norme internationale [ISO 00] est basé en grande partie sur la proposition de LOTOS temps-réel ET-LOTOS [LÉO 97, LÉO 98a, LÉO 98b], sur la proposition de LOTOS avec types de données fonctionnels [JEF 95a, JEF 96a], et sur l'ajout d'exceptions [GAR 96b]. Un grand nombre de constructions de langage, notamment les constructions impératives, sont basées sur le langage proposé dans [GAR 96a]. La première version intégrée d'un noyau du langage avec sa sémantique a été proposée dans [JEF 96b].

2. Le langage E-LOTOS de base

2.1. *Déclarations*

Une spécification dans le langage de base comprend un ensemble de *déclarations*. Ces déclarations peuvent être structurées au niveau des modules (Cf. section 3).

Les déclarations sont de trois sortes : déclarations de *type*, déclarations de fonctions (*function*), et déclarations de processus (*process*). Dans le langage de base, tous les identificateurs de types, de constructeurs, de fonctions et de processus doivent être uniques : la gestion de la surcharge des noms est faite au niveau du langage de modules.

Déclaration de type

Une déclaration de type est soit la définition d'un *type synonyme*, soit la déclaration d'un *type de donnée*.

Un type synonyme déclare un nouvel identificateur pour un type existant :

type code **renames nat endtype**

Dans ce cas, code et **nat** sont synonymes. Ils sont interchangeables. Nous pouvons utiliser code et **nat** comme un seul type (par exemple, toute fonction requérant un code acceptera un **nat**). Plus généralement, l'égalité des types est *structurelle* en E-LOTOS, et non par *nom*.

Par ailleurs, nous pouvons déclarer une liste d'entiers comme un type de donnée récursif :

```
type intlist is
    nil | cons(int, intlist)
endtype
```

Les déclarations de types de données définissent de nouveaux types, en énumérant tous les *constructeurs* de ce type. Puisqu'il peut y avoir plus d'un constructeur, nous pouvons définir des types *union*, comme suit :

```
type pdu is
  send(packet, bit) | ack(bit)
endtype
```

Notons encore que le langage de base ne fournit *aucun* mécanisme de définition de type paramétré. Ceci est du ressort du système de modules. De même, le système de modules offre une riche collection de types prédéfinis (**bool**, **nat**, **int**, **rational**, **float**, **char**, **string**), ainsi que des schémas de définition pour les types énumérés, enregistrement, ensembles et listes. (Cf. section 3.6).

Déclarations de fonctions

Une déclaration de fonction définit une nouvelle fonction, qui peut être utilisée dans les expressions de donnée. Par exemple :

```
function reflect (p:point) : point is
  (x=>p.y,y=>p.x)
endfunc
```

Les paramètres de fonctions sont donnés par des listes de variables typées. Une fonction peut avoir plusieurs paramètres et peut renvoyer une liste de résultats (dans un type enregistrement). Par exemple (nous définirons les détails plus loin) :

```
type PairOfIntLists is (intlist,intlist) endtype
function partition (x:int,xs:intlist) : PairOfIntLists is
  var
    less:intlist := (* tous les xs inférieurs à x *) ...,
    gtr:intlist:= (* tous les xs supérieurs à x *) ...
  in
    (less,gtr)
  endvar
endfunc
```

En utilisant le prefixage par '?' pour marquer une *occurrence de liaison* (affectation) de variables, nous pouvons appeler cette fonction comme dans l'exemple suivant :

```
function quicksort (xs:intlist) :intlist is
  case xs is
       nil ->nil
    | cons(?y,?ys) ->
        var  l:intlist, g:intlist
        in
          (?l,?g) := partition (y,ys);
          append (quicksort (l),cons (y,quicksort (g)))
        endvar
  endcase
endfunc
```

Ce style de fonction est très commun, de telle sorte que le langage fournit une notation conviviale pour l'exprimer, en utilisant des paramètres **in** et **out**. Par exemple, la fonction 'partition' aurait pu être écrite comme suit :

```
function partition (in x:int, in xs:intlist,
                    out less:intlist, out gtr:intlist) is
  ?less := (* tous les xs inférieurs à x *) ...;
  ?gtr := (* tous les xs supérieurs à x *) ...
endfunc
```

et ensuite utilisée dans 'quicksort' ainsi :

```
partition (y,ys,?l,?g);
```

Les fonctions peuvent générer des exceptions qui devront être déclarées dans l'entête de la fonction comme dans l'exemple suivant :

```
function hd (xs:intlist) : int raises [Hd] is
  case xs is
       nil -> raise Hd
    | cons(?x,any:intlist) -> x
  endcase
endfunc
```

Quand une telle fonction est appelée, l'exception 'Hd' doit être instanciée. Par exemple, l'expression suivante va générer l'exception 'Foo' :

```
hd (nil) [Foo]
```

Les exceptions peuvent porter plusieurs valeurs dont les types doivent être spécifiés à la déclaration de l'exception. Par exemple :

```
function foo () raises [Foo:(string)] is
  raise Foo("Hello world")
endfunc
```

Toute exception déclarée sans type se voit attribuer par défault le type enregistrement vide `()` (Cf. section 2.2).

Rappelons que les déclarations de fonctions ne constituent qu'une sous-classe des déclarations de processus.

Déclarations de processus

Les déclarations de processus sont très similaires aux déclarations de fonctions, car elles contiennent entre autres une liste de paramètres valeurs (**in** et **out**) et une liste d'exceptions typées que le processus peut générer. Cependant, il existe deux différences importantes entre fonction et processus : les processus peuvent communiquer par des portes et peuvent avoir un comportement temps-réel. Par exemple, un simple compteur est défini comme suit :

```
process Counter [up,down] is
  up; (down ||| Counter [up,down])
endproc
```

Les comportements de processus sont discutés en détail dans la section 2.4.

2.2. *Typage*

Expressions de types

Nous avons déjà rencontré quelques expressions de types. Par exemple :

- Le type de donnée 'intlist' et le type synonyme 'code' sont tous deux des *identificateurs de types*. Les types primitifs **int** et **string** sont également des identificateurs de types.
- Le type `(x=>float,y=>float)` est un *type enregistrement* muni des *champs* 'x' et 'y'.
- Le type `(int,intlist)` est un *type paire* : en fait, c'est une notation compacte pour le type enregistrement `($1=>int,$2=>intlist)`, que l'on utilise lorsque les noms des champs sont sans importance. Ils ont alors implicitement les noms '$i', où i est la position du champ dans l'enregistrement.

Les types enregistrement sont *extensibles*. Par exemple, `(name=>string,etc)` est un type enregistrement ayant au moins un champ de type **string**, mais qui peut être étendu de façon à contenir d'autres champs.

En plus des identificateurs de types et des types enregistrement, il existe deux types spéciaux :

- Le type sans valeurs, **none**, utilisé pour donner une fonctionnalité aux processus qui ne se terminent pas tels que **stop** ou 'Counter'. Le type **none** est le type par défaut d'un processus n'ayant aucun paramètre out.
- Le type universel, **any**, qui est un supertype de tous les autres types.

Sous-typage

Le langage de base permet le *sous-typage*. Le sous-typage sur les enregistrements est disponible par construction dans la sémantique : le type enregistrement **(etc)** est un supertype de tous les types enregistrement. Par exemple, le type (name=>**string**,**etc**) est un enregistrement avec au moins le champ 'name' de type **string**. Ce type enregistrement peut être étendu à plusieurs sous-types tels que (name=>**string**,age=>**int**,**etc**) ou (name=>**string**,age=>**int**). Remarquons la différence entre ces deux derniers types : le premier peut à son tour être étendu, alors que le second ne le peut plus.

Le type **none** est le type le plus spécialisé, et **any** est le plus général. L'ensemble des types forme donc un treillis. Puisqu'un type enregistrement ayant un champ de type **none** ne peut avoir aucune valeur (vu qu'un de ses champs ne peut en avoir aucune), nous pouvons l'identifier à **none**. Par exemple, la paire **(none,int)** n'a pas de valeur et est donc équivalente au type **none**. Ceci signifie aussi que le type enregistrement à un champ **(none)** est le type enregistrement le plus spécialisé, et que **(etc)** est le type enregistrement le plus général.

Par exemple, **stop** est un comportement de type **exit (none)**. Ce qui signifie qu'il ne se termine pas. Puisque **(none)** est le moins général des types enregistrement, nous pouvons utiliser **stop** à tout endroit où un processus requiert un type enregistrement quelconque.

De façon similaire, si G est une porte de type **(etc)**, nous pouvons communiquer des valeurs de tout type enregistrement par G. Le fait que le type par défaut d'une porte soit **(etc)**, permet ainsi aisément d'être compatible avec LOTOS où les portes sont non typées.

2.3. *Expressions de données*

Contrairement à LOTOS, E-LOTOS considère les fonctions comme des processus particuliers qui ne communiquent pas et s'exécutent de façon instanta-

née. Le langage des expressions est donc très similaire au langage des comportements, et partage avec celui-ci de nombreux aspects tels que le filtrage des valeurs (*pattern-matching*), la génération et la capture d'exceptions, les constructions impératives.

Formes normales

Une *forme normale* est une expression de donnée qui ne peut plus être réduite. Par exemple $1+1$ n'est pas une forme normale, mais 2 l'est. Une forme normale est une des formes suivantes :

- Une constante primitive, telle que `"Hello world"` ou 2, appartenant à un des types de données primitifs.
- Une variable, telle que 'x' ou 'gtr'.
- Un enregistrement de formes normales, comme `(x=>`1.5`,y=>` $-$ 3.14`)`, `()` ou `(5,nil())` (ce dernier n'est qu'une notation compacte pour `($1=>5,$2=>nil())`).
- Un constructeur appliqué à une forme normale, comme 'nil`()`' ou 'cons`(5,nil())`'.

Nous utiliserons N pour représenter une forme normale, et `(`RN`)` pour un enregistrement de formes normales.

Filtrage

Le langage des expressions inclut une opération **case**, qui permet le branchement selon la valeur d'une expression. Par exemple, nous pouvons obtenir la tête d'une liste comme suit :

```
case xs is
     nil -> raise Hd
   | cons(?x,any:list) -> x
endcase
```

Cette opération **case** prend en entrée une expression (ici 'xs') et, en fonction de sa valeur, effectue le branchement vers la possibilité dont le filtre (*pattern*) convient à la valeur. Dans l'exemple, si la valeur de 'xs' est la liste vide, le premier filtre convient et l'exception 'Hd' est générée. Si 'xs' n'est pas vide, c'est le second filtre qui convient, et 'x' sera *liée à* la tête de liste, et renvoyée comme résultat de la fonction.

Les filtres de l'expression **case** sont évalués dans l'ordre, du premier au dernier jusqu'à ce que l'un convienne. Si aucun filtre ne convient (ce qui ne peut être le cas dans l'exemple ci-dessus), une exception spéciale 'Match' est générée.

Remarquons que 'cons(?x,**any**:list)' est un filtre structuré. Au plus haut niveau, nous trouvons le constructeur de listes 'cons', appliqué à un filtre d'enregistrement qui inclut les filtres élémentaires '?x' et '**any**:list'. Pour qu'une liste convienne à ce filtre, elle doit avoir la forme 'cons(hd,tl)'.

Nous autorisons aussi la présence des expressions de données dans les filtres. L'usage le plus fréquent correspond à des filtres constants, par exemple :

```
case x is
     !0 -> "zero"
   | any:int -> "nonzero"
endcase
```

L'exemple suivant illustre l'intérêt d'avoir une expression comme filtre. Il montre comment tester si une liste est un palindrome en utilisant une fonction qui inverse une liste :

```
case xs is
     !reverse(xs) -> "palindrome"
   | any:list -> "nonpalindrome"
endcase
```

Nous verrons dans la section 2.4 que cette construction est aussi particulièrement utile lors des communications entre processus.

Les filtres peuvent être typés explicitement, ce qui est utile en présence de sous-typage. Par exemple, si **nat** était un sous-type de **int**, nous pourrions construire une instruction **case** pour décider si une valeur est un naturel ou non :

```
case x:int is
     any:nat -> "natural"
   | any:int -> "integer"
endcase
```

A nouveau, l'intérêt principal est au niveau des communications.

Un filtre peut avoir l'une des formes suivantes :

- Une variable à lier, telle que '?x'.
- Une expression comme '!0' ou '!reverse(xs)'.
- Le filtre joker typé '**any**:T'.

- Un filtre d'enregistrement, comme `(x=>?px,y=>?py)`, `()`, ou `(?x,any:T)`
  (ce dernier est juste une notation compacte pour `($1=>?x,$2=>any:T)`).
- Un filtre d'enregistrement extensible, comme `(x=>?px,etc)`, `(etc)`, ou `(?x,etc)` où **etc** est un filtre qui peut correspondre à la liste de tous les autres champs. Remarquons la différence entre `(?x,any:T)` et `(?x,etc)` : seul un enregistrement à deux champs pourra correspondre au premier, alors que tout enregistrement ayant au moins un champ pourra correspondre au second.
- Un filtre enregistrement avec une clause **as** qui lie une partie de l'enregistrement, comme dans : `(?all as ?x,etc)` ou `(?x,?all as etc)`.
- Un constructeur appliqué au filtre, comme 'nil' ou 'cons`(?x,any:list)`'.
- Un filtre explicitement typé, comme '`?y:int`'.
- Un filtre conditionnel, comme '`?y:int [y < 10]`' qui accepte tout entier inférieur à 10.

L'opérateur **case** permet aisément de construire d'autres opérateurs, comme l'instruction **if**. L'expression suivante :

$$\textbf{if } E \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ endif}$$

peut être transcrite comme suit :

$$\begin{array}{l}\textbf{case } E \textbf{ is}\\ \quad \text{true -> } E_1\\ \quad \textbf{| any:bool -> } E_2\\ \textbf{endcase}\end{array}$$

Idem pour les instructions **elsif** :

$$\textbf{if } E_1 \textbf{ then } E_2 \textbf{ elsif } E_3 \textbf{ then } E_4 \textbf{ else } E_5 \textbf{ endif}$$

peut être transcrit comme suit :

$$\begin{array}{l}\textbf{if } E_1 \textbf{ then } E_2\\ \textbf{else if } E_3 \textbf{ then } E_4 \textbf{ else } E_5 \textbf{ endif}\\ \textbf{endif}\end{array}$$

Exceptions

Les expressions peuvent générer des exceptions qui signalent certaines erreurs. Par exemple, dans la fonction 'hd', une exception sera générée si l'on tente d'accéder à la tête d'une liste vide.

Les exceptions se propagent jusqu'au niveau le plus haut de la spécification, sauf si elles sont capturées par un *gestionnaire d'exceptions* décrit au moyen de l'opérateur **trap**. Par exemple, si nous déclarons la fonction suivante :

```
function hd0 (xs:intlist) : int is
  trap
    exception Hd is 0 endexn
  in
    hd (xs) [Hd]
  endtrap
endfunc
```

alors 'hd0 (cons(a,as))' renvoye 'a', et 'hd0 (nil)' renvoye 0, puisque l'exception 'Hd' générée par 'hd' est capturée par le gestionnaire.

Les exceptions sont typées par le type des valeurs qu'elle porte. Ainsi, le traitement de l'exception peut dependre de ces valeurs :

```
trap
  exception Error (code:int) is
    case code is
        !0 -> "minor error"
      | !1 -> "major error"
      | any:int -> raise Unknown (code)
    endcase
  endexn
in ...
endtrap
```

Nous pouvons aussi déclarer plus d'une exception dans un seul gestionnaire :

```
trap
  exception Foo is E1 endexn
  exception Bar is E2 endexn
in E
endtrap
```

Remarquons que Foo et Bar ne sont capturées que dans E, *et non* dans E_1 ni E_2. Ainsi, si E génère 'Foo' ou 'Bar', le gestionnaire les capture, mais si E_1 ou E_2 génèrent 'Foo' ou 'Bar', ces exceptions ne seront pas capturées.

De plus, nous pouvons écrire un gestionnaire pour la terminaison réussie d'une expression, comme dans l'exemple suivant :

```
trap
  exception ParseError is 0 endexn
  exit (x:string) is string2int (x) [ParseError] endexit
in E
endtrap
```

Cette construction est utile dans le cas où nous voulons que toute exception 'ParseError' générée par E soit capturée, mais *pas* les exceptions 'ParseError' générées par 'string2int'. Il ne serait pas possible d'écrire cette expression sans cette capacité à capturer une terminaison réussie. Aucune des deux solutions ci-dessous ne conviendrait. La première n'est pas bien typée :

```
string2int (
  trap exception ParseError is 0 endexn
  in E
  endtrap
) [ParseError]
```

et la seconde capture l'exception 'ParseError' générée par 'string2int' :

```
trap exception ParseError is 0 endexn
in string2int (E) [ParseError]
endtrap
```

L'opérateur **trap** joue donc un double rôle. Il capture les exceptions, mais il sert aussi à les déclarer. Ce qui signifie qu'une exception ne peut s'échapper de sa portée, contrairement à d'autres langages comme SML [MIL 90] où la déclaration et la capture d'une exception sont séparées.

Remarquons que le seul moyen d'observer une exception est de la capturer. Deux processus ne peuvent se synchroniser sur une exception.

Constructions impératives

Le langage des expressions de données est fonctionnel, mais dispose d'expressions qui imitent un langage impératif muni de (ré)affectations. L'expression suivante :

```
?x := 0; ?y := "hello world";
```

est équivalente au comportement :

```
(x=>0,y=>"hello world")
```

L'expression impérative la plus simple est l'affectation P `:=` E, où P est un filtre irréfutable (c'est-à-dire, qui filtre toute valeur) et E une expression. Par exemple :

```
?x := 4
```

Nous avons vu précédemment que nous autorisions les paramètres **out** comme une notation compacte pour des affectations. Par exemple :

```
partition (y,ys,?l,?g)
```

est synonyme de :

```
(?l,?g) := partition (y,ys)
```

Il existe un opérateur de composition séquentielle dont la syntaxe est E_1 **;** E_2. Il ressemble à l'opérateur d'activation (‘**>>** ’) de LOTOS, car il combine deux expressions, mais sa sémantique est un peu différente, car il ne génère aucune action interne **i**.

L'opérateur **var** est utilisé pour restreindre la portée des variables, avec la syntaxe ‘**var** ILV **in** E **endvar**’, où ILV est une liste de variables typées. Par exemple :

```
var x:int
in
  ?x:=E; x * x
endvar
```

a la même sémantique que $E * E$. Il est aussi possible d'initialiser les variables locales. Nous aurions pu écrire :

```
var x:int:=E
in
  x * x
endvar
```

Un opérateur d'itération (de boucle) est également présent dans le langage. Il permet d'exprimer des processus récursifs sans nommer explicitement le processus, ce qui est un avantage par rapport à LOTOS. Les boucles peuvent avoir des variables locales. Ces variables peuvent être initialisées, et doivent être assignées à chaque itération de la boucle. Une boucle peut être interrompue par la commande **break**, qui peut porter une valeur. L'exemple suivant montre une fonction impérative qui additionne les entiers d'une liste :

```
function sum (xs:intlist) : int is
  var ys:intlist:=xs, total:int:=0
  in
    loop
      case ys is
          nil -> break (total)
        | cons(?z,?zs) -> ?total := total + z; ?ys := zs
      endcase
    endloop
  endvar
endfunc
```

La boucle avec interruption se traduit syntaxiquement à l'aide des opérateurs **trap** et **loop**. Par exemple, la boucle précédente est une description compacte ayant la même sémantique que :

```
trap
  exception Inner (x:int) is x endexn
in
  loop
    var ys:intlist:=xs, total:int:=0
    in
      case ys is
           nil -> raise Inner (total)
        | cons(?z,?zs) -> ?total := total + z; ?ys := zs
      endcase
    endvar
  endloop
endtrap
```

Les boucles peuvent être nommées, de telle sorte que l'on puisse sortir d'une boucle qui n'est pas la plus interne :

```
loop fred in ...
  loop janet in ...
    if b then break fred ...
```

En outre, le langage possède les boucles classiques **while** et **for**, qui sont des notations compactes définies à partir de **loop**. La syntaxe de la boucle **while** est classique ; l'exemple suivant montre la syntaxe pour la boucle **for**.

```
function fact (n:int) : int raises [OutOfRange] is
  if n < 0 then raise OutOfRange endif;
  var x:int,res:int:= 1 in
    for x:= 2 while x <= n by ?x:= x+1 do
      res:= ?res * x
    endfor
  endvar
endfunc
```

2.4. *Expressions de comportement*

En ce qui concerne les comportements, les différences principales entre LOTOS et le langage E-LOTOS de base sont les suivantes :

- Les actions sont devenues des comportements particuliers, ce qui a comme consequence l'unification des deux formes de composition séquentielle (préfixage par une action et activation).

- Des nouvelles constructions sont ajoutées comme le filtrage des valeurs, les exceptions, l'affectation, le temps et quelques autres opérateurs (comme le renommage explicite).

Le langage des comportements peut être vu comme une extension du langage de données selon deux axes : la communication (ou synchronisation) et le temps.

Communication

Les processus communiquent par des *portes*. Le plus simple processus communicant est celui qui se limite à une synchronisation sur la porte G : ce qui s'écrit simplement G. De telles synchronisations peuvent alors se composer séquentiellement, comme dans l'exemple suivant où les actions 'inp' et 'outp' alternent :

```
loop
  inp; outp
endloop
```

Les processus peuvent aussi émettre et recevoir des données par les portes, comme dans l'exemple d'un buffer à une place pouvant contenir des nombres entiers :

```
loop
  var x:int
  in
    inp ?x; outp !x
  endvar
endloop
```

Ici la variable 'x' est *liée* lors de la communication par la porte 'inp', et est *libre* lors de la communication par la porte 'outp'. Le comportement résultant copie des entiers de la porte 'inp' vers la porte 'outp'.

Lors d'une synchronisation sur une porte, on peut spécifier un filtre quelconque pour les valeurs reçues. Par exemple, le comportement :

```
G(age=>!28,name=>?na,
   address=>(number=>?no,street=>!"Acacia Ave",etc))
```

se synchronise sur G avec toute personne agée de 28 ans et habitant Acacia Avenue, et liera les variables 'na' et 'no' respectivement au nom et au numéro. Cet usage des filtres dans les communications justifie les notations '`?`' et '`!`' dans les filtres en général.

On peut aussi spécifier un *prédicat de sélection* indiquant si une synchronisation doit être autorisée. Par exemple, pour sélectionner un quadragénaire résidant à Acacia Avenue, on écrirait :

```
G(age=>?a,name=>?na,address=>(street=>!"Acacia Ave",etc))
   [40 ≤ a andalso a ≤ 49]
```

Nous avons vu (Cf. processus 'Buffer') que les déclarations de processus spécifient, en plus des paramètres valeur, les portes de communication du processus. Ces portes peuvent être typées. Par défaut, une porte a le type **(etc)**, et peut donc communiquer des enregistrements de tout type enregistrement :

```
process OverloadingExample [overloaded] (out x:int, out y:bool) is
  overloaded(?x:int);
  overloaded(?y:bool)
endproc
```

La première communication sur la porte doit se faire avec un (enregistrement ayant un champ) entier, et la seconde avec un booléen. Le type de la porte autorise les deux dans ce cas, vu qu'il vaut **(etc)**.

Nous pouvons utiliser les filtres **as** pour mettre une variable en correspondance avec un sous-ensemble (ou l'ensemble) des champs d'une communication. C'est particulièrement utile lorsque le type de la porte est un enregistrement extensible. Par exemple, un simple routeur capable de manipuler des paquets contenant des données de tout type peut s'écrire :

```
type S is (de=>dest,etc) endtype
type AnyRecord is (etc) endtype
process Router [inp:S,left,right] is
  var destination:dest, data:AnyRecord
  in
    inp(de=>?destination,?data as etc);
    case destination is
        L -> left!data
      | R -> right!data
    endcase;
    Router [inp, left, right]
  endvar
endproc
```

Parallélisme

E-LOTOS herite de LOTOS les opérateurs classiques de composition parallèle. Par exemple, deux processus qui sont forcés de se synchroniser lors de toute communication peuvent s'écrire :

```
   G(address=>(number=>?no,street=>!"Acacia Ave",etc),etc)
|| G(age=>!28,name=>?na,address=>any:addrType)
```

Puisque les deux comportements sont forcés de se synchroniser sur la porte G, ce système a la même sémantique que :

```
G(age=>!28,name=>?na,
  address=>(number=>?no,street=>!"Acacia Ave",etc))
```

Les communications peuvent être bidirectionnelles lors d'une synchronisation. Par exemple :

```
    G(age=>!28,name=>?na,etc);B1
||  G(age=>?a,name=>!"Fred",etc);B2
```

a la même sémantique que :

```
G(age=>!28,name=>!"Fred",etc);
(?na:="Fred"; B1) || (?a:=28; B2)
```

Les processus parallèles doivent se synchroniser lors d'une terminaison. Par exemple, le comportement suivant se termine immédiatement après que les variables 'x' et 'y' aient été toutes les deux affectées :

```
?x:=1 || ?y:=2
```

Deux comportements qui ne se synchronisent pas (sauf pour la terminaison) sont les suivants :

```
overloaded(?x:int) ||| overloaded(?y:bool)
```

Ce processus va communiquer deux fois via la porte 'overloaded' : une fois en acceptant un entier, et une autre fois en acceptant un booléen, sans que l'ordre ne soit précisé. Le processus se termine après ces deux opérations. Il a la même sémantique que :

```
    overloaded(?x:int); overloaded(?y:bool)
[]  overloaded(?y:bool); overloaded(?x:int)
```

Remarquons que les variables liées pas les processus parallèles sont toutes les variables liées par les composants, et qu'il n'y a pas de possibilité de communication par partage de variables.

Opérateur de parallélisme généralisé

E-LOTOS dispose d'un opérateur de composition parallèle qui permet la synchonisation explicite d'un sous-ensemble de n processus parmi l'ensemble des processus communicants :

par $G_1\#n_1, ..., G_p\#n_p$ **in**
  $[\Gamma_1]$ **for** B_1
|| ...
|| $[\Gamma_n]$ **for** B_n
endpar

Cet opérateur signifie que si un processus B_i peut exécuter une action sur la porte G, et que cette action est spécifiée dans la liste Γ_i (la *liste de synchronisation*), alors cette action doit être synchronisée avec les actions des autres composants de la façon suivante :

- si G est spécifiée dans la *liste des degrés* ($G_1\#n_1, ..., G_p\#n_p$) avec le dégré n, alors B_i doit se synchroniser par G avec n-1 autres composants qui ont G dans leur liste de synchronisation ;
- si G n'apparaît pas dans la liste des degrés, alors B_i doit se synchroniser avec tous les autres composants ayant G dans leur liste de synchronisation.

Par ailleurs, si G n'est pas dans la liste Γ_i, B_i peut exécuter G seul, sans synchronisation avec les autres composants B_j.

Temps

Les processus ont des comportements temps réel, grâce aux trois constructions suivantes :

- Un type 'time', muni d'opérations d'addition et de comparaison,
- Un opérateur **wait**, introduisant des délais, et
- Une extension de l'opérateur de communication, qui le rend sensible au temps qui passe.

Le type de donnée 'time' est un ordre total avec addition. Nous noterons d une valeur quelconque de type 'time'.

L'opérateur de délai est simplement noté **wait**(d). C'est un processus qui attend d unités de temps puis se termine. Par exemple, un processus qui communique par la porte G à chaque unité de temps s'écrira :

```
loop
  G; wait(1)
endloop
```

Nous pouvons créer un délai donné par la valeur d'une expression **wait**(*E*), comme dans l'exemple suivant :

```
loop
  var x:time
  in
    G ?x; wait(x)
  endvar
endloop
```

De même, nous pouvons introduire un délai non déterministe comme suit :

```
loop
  var x:time
  in
    G; ?x := any time; wait(x)
  endvar
endloop
```

Les communications peuvent dépendre du temps si l'on ajoute une annotation `@`*P*, qui fait correspondre le filtre *P* au temps auquel la communication se produit. Ce temps est mesuré à partir de l'instant d'activation de la communication. Par exemple :

$$G\ \texttt{?x:int@?t[t} < 3\texttt{]}$$

est un comportement qui accepte un entier (qui sera affecté à la variable 'x'), pour autant que moins de 3 unités de temps se soient écoulées. Par contre,

$$G\ \texttt{?x:int@!3}$$

est très semblable, mais l'action ne peut se produire qu'à l'instant 3, car le filtre est réduit à un filtre de valeur définie`!`3. Ce comportement a la même sémantique que :

```
var t:time
in
  G ?x:int@?t[t = 3]
endvar
```

Ces constructions temporelles sont directement inspirées de ET-LOTOS [LÉO 97], mais ont été adaptées aux nouveaux paradigmes du langage, comme : le fait qu'une action soit un comportement, que la composition séquentielle ne génère plus d'action interne **i**, la présence du filtrage de valeurs, et la présence d'exceptions.

Urgence

L'*urgence* est un concept important : un comportement est urgent s'il ne peut être retardé, par exemple s'il y a un calcul qui doit être effectué immédiatement. La composition séquentielle est urgente. Cela signifie que lorsque le premier comportement se termine, le contrôle est passé au comportement suivant sans attendre. Considérons le processus suivant :

```
loop
   (loop tick endloop [> wait(1));
   (loop tock endloop [> wait(1))
endloop
```

Il va exécuter un nombre quelconque d'actions 'tick' pendant le premier intervalle de temps, puis au temps 1, le chien de garde l'interrompt et le contrôle est passé à la deuxième boucle où le processus pourra exécuter un nombre quelconque d'actions 'tock' jusqu'au temps 2, etc. Le passage d'une boucle de 'tick's à une boucle de 'tock's (et inversement) est instantané, de sorte qu'on est sûr qu'aucun 'tick' (respectivement 'tock') ne sera exécuté pendant un intervalle pair (respectivement impair).

En E-LOTOS, les actions suivantes sont urgentes :

- l'action interne (**i**), qu'elle soit écrite explicitement ou qu'elle résulte d'une opération **hide**,
- la génération d'une exception (X) et
- l'action de termination (δ).

Toutes ces actions se produisent immédiatement, sauf lorsqu'il s'agit d'une action terminaison qui doit se synchroniser avec la terminaison simultanée des comportements parallèles. Par exemple, le comportement suivant se terminera au temps 2 :

```
wait(1) ; exit || wait(2) ; exit
```

La sémantique de l'urgence des exceptions est essentiellement celle du modèle des signaux de Timed CSP [DAV 92].

Abstraction

La syntaxe de l'opérateur **hide** est celle de LOTOS où l'on a ajouté le typage des portes (déclarées). Dans l'exemple suivant :

```
hide mid:int in
  Buffer [inp,mid] || Buffer [mid,outp]
endhide
```

une nouvelle porte 'mid' est déclarée (elle ne peut échanger que des entiers ou d'éventuels sous-types) et est cachée à l'environnement par transformation en actions internes **i**. Cet opérateur préserve la propriété d'urgence de tous les **i** ainsi créés, et induit donc l'urgence sur les synchronisations cachées. Ce qui signifie que l'on exprime qu'une synchronisation se produit dès que tous les processus impliqués le permettent. La synchronisation sur une terminaison réussie était déjà de ce type. Par exemple, le comportement suivant :

hide G **in**
  wait(1) ; G ; B_1 || **wait**(2) ; G ; B_2
endhide

a la même sémantique que :

wait(2) ; **i** ;
hide G **in**
  B_1 || B_2
endhide

L'action G se produit après 2 unités de temps, ce qui correspond au moment où les deux processus peuvent exécuter G. Le comportement suivant :

hide G **in**
  G@?t[t $\geq$ 3] ; B
endhide

a la même sémantique que :

wait(3) ; ?t:=3 ; **i** ;
hide G **in** B **endhide**

A nouveau, le premier instant où G est possible est après 3 unités de temps.

Le comportement suivant :

hide G **in**
  G@?t[t $>$ 3] ; B
endhide

a deux sémantiques possibles selon que le type 'time' représente un temps dense ou discret. Si 'time' est un synonyme de nombres naturels (modèle de temps discret), le comportement a la sémantique suivante :

wait(4) ; ?t:=4 ; **i** ;
hide G **in** B **endhide**

parce que 4 est le plus petit nombre naturel strictement supérieur à 3. Par contre, si 'time' est synonyme de nombres rationnels (modèle de temps dense), le comportement a la même sémantique que

wait(3) ; **block**

La raison pour laquelle ce processus ne peut progresser au delà de l'instant 3 (et ne peut même pas exécuter l'action cachée G) est qu'il n'existe pas de plus petit nombre rationnel strictement supérieur à 3.

Devoir cacher les synchronisations pour qu'elles se produisent dès que possible est parfois critiqué, car il existe des cas où l'on souhaiterait continuer à observer ces portes. Le problème réside dans l'interprétation du terme 'observation'. Observer nécessite une interaction, et toute interaction peut créer une interférence non désirée. L'idéal serait de pouvoir rendre une (inter)action visible à l'environnement sans autoriser celui-ci à interférer. Il existe une solution à ce problème. Il suffit de générer une exception (encore appélée 'signal' dans ce contexte) immédiatement après l'occurrence d'une (inter)action cachée. Considérons deux comportements, 'Producer' et 'Consumer', qui souhaitent se synchroniser sur l'action 'sync' dès qu'ils sont tous deux prêts à le faire. Nous ajoutons un comportement spécial de surveillance ('Monitoring') de cette action qui se synchronise avec eux et envoie un signal juste après l'occurrence de 'sync' :

Producer ≡ **loop** ?t1:= **any** time; **wait**(t1); sync **endloop**
Consumer ≡ **loop** sync; ?t2:= **any** time; **wait**(t2) **endloop**
Monitoring ≡ **loop** sync; **signal** yes **endloop**
System ≡ **hide** sync **in** (Producer || Consumer || Monitoring)

L'opérateur **signal** est quasi identique à **raise** sauf qu'il permet au comportement de continuer son exécution après la génération de l'exception (si celle-ci n'est pas capturée par un gestionnaire) : **raise** X est en fait une notation compacte pour l'expression **signal** X ; **block**.

Suspension/Redémarrage

Cet opérateur, proposé dans [HER 98] est une extension de l'opérateur de désactivation de LOTOS, qui permet de reprendre l'exécution du processus suspendu lorsque le processus d'interruption le décide. Sémantiquement, la reprise du processus interrompu s'effectue par le biais d'une exception spécifiée comme paramètre de l'opérateur. Par exemple, dans le comportement suivant :

wait(4) ; B_1 [X > **wait**(1) ; i; **wait**(2) ; **raise**(X)

le comportement à gauche du '[X >' est suspendu à toutes les unités de temps par l'action interne du processus de droite. La suspension dure 2 unités de temps au bout desquelles le processus de gauche redémarre grâce à l'exception X. Quand un processus est suspendu, il ne peut exécuter aucune action et il ne 'vieillit' pas non plus, c'est-à-dire que le temps ne s'écoule pas pour lui. Dans l'exemple, il en résulte que le processus est redémarré dans l'état suivant après la première suspension :

$$\textbf{wait}(3)\,;\ B_1\ \ [\mathrm{X}>\ \ \textbf{wait}(1)\,;\ \mathrm{i};\ \textbf{wait}(2)\,;\ \textbf{raise}(\mathrm{X})$$

et non pas dans l'état

$$\textbf{wait}(1)\,;\ B_1\ \ [\mathrm{X}>\ \ \textbf{wait}(1)\,;\ \mathrm{i};\ \textbf{wait}(2)\,;\ \textbf{raise}(\mathrm{X}).$$

Le comportement de droite est toujours réactivé dans son état initial lorsque le processus suspendu redémarre, ce qui signifie que les deux comportements suivants ont la même sémantique :

$$\begin{array}{l} B_1\ \ [\mathrm{X}>\ \ B_2\,;\ \textbf{raise}(\mathrm{X}) \\ B_1\ \ [\mathrm{X}>\ \ B_2\,;\ \textbf{signal}(\mathrm{X})\,;\ B_3 \end{array}$$

Cet opérateur simple peut être utilisé pour spécifier des mécanismes d'interruption plus complexes où, par exemple, un comportement peut en suspendre plusieurs autres. Considérons le comportement suivant :

$$\begin{array}{l} (B_1\ [X_1>\ G_1\,;\ G_1'\,;\textbf{raise}(X_1)\ \ |||\ \ B_2\ [X_2>\ G_2\,;\ G_2'\,;\textbf{raise}(X_2)) \\ |[G_1, G_2, G_1', G_2']| \\ B_3 \end{array}$$

Le comportement B_3 contrôle B_1 et B_2 via les portes G_i et G_i'. B_3 peut suspendre B_1 par la porte G_1 et le redémarrer par la porte G_1'. Ainsi, il est possible de spécifier des mécanismes d'interruption plus complexes, comme des ordonnanceurs temps-réels.

Remarquons que l'opérateur de désactivation de LOTOS est un cas particulier de cet opérateur suspension/redémarrage où aucune exception de reprise n'est spécifiée.

Renommage

Un opérateur de renommage explicite des actions a été introduit dans le langage. Il permet de renommer des actions observables et des exceptions, sans toutefois permettre de transformer une action de communication en exception ou inversement.

Renommer une action observable peut apporter plus d'expressivité que l'on ne pourrait le croire au premier abord. Le renommage permet plus que le simple changement de nom de porte. Il permet aussi de changer la structure des événements se produisant à une porte, voire même de fusionner deux portes ou d'éclater une porte.

La forme la plus simple de renommage modifie le nom de la porte :

```
rename gate
   G (x=>?i:int) is G'(x=>!i)
in
  B
endren
```

Remarquons la similitude de syntaxe entre le renommage, la déclaration de fonction et la capture d'exceptions. Cette forme de renommage est si commune que E-LOTOS en propose une notation compacte :

```
rename gate
  G (x=>int) is G'
in
  B
endren
```

Nous pouvons abstraire un champ dans le message echangé à une porte :

```
rename gate
  G (x=>?i:int,y=>any:bool) is G'(!i)
in
  B
endren
```

Nous pouvons ajouter un champ au message echangé à une porte :

```
rename gate
   G (x=>?i:int) is G'(x=>!i,y=>!true)
in
   B
endren
```

Nous pouvons fusionner deux portes G' et G'' en une nouvelle porte G :

```
rename
  gate G' (x=>?i:int) is G(x=>!i,y=>!true)
  gate G'' (x=>?i:int) is G(x=>!i,y=>!false)
in
  B
endren
```

Enfin, nous pouvons aussi éclater une porte G en deux portes G_1 et G_2 :

```
rename
  gate G (x=>?i:int,y=>!1) is G1(!i)
  gate G (x=>?i:int,y=>!2) is G2(!i)
in
  B
endren
```

Les exceptions peuvent être renommées de façon similaire.

Constructions impératives dans les comportements

Comme pour les expressions, certaines constructions de nature impérative permettent de simplifier l'écriture des processus. L'exemple suivant en donne une illustration :

```
process protocol [down:packet,up] is
  var code:packet, data:AnyRecord
  in
    down(de=>?code,?data as etc) ;
    while code<>disconnect do
      up!data ; down(de=>?code,?data as etc)
    endwhile
  endvar
endproc
```

2.5. *Aspects sémantiques*

La sémantique statique du langage de base est construite à partir de jugements tels que $\mathcal{C} \vdash E \Rightarrow$ **exit** (T). Un tel jugement signifie 'dans le contexte des déclarations $\mathcal{C}$, l'expression E a pour type T'. Par exemple :

$1 \Rightarrow$ **float**, $x \Rightarrow$ **float**, $/ \Rightarrow$ **(float, float) → exit (float)** $\vdash 1/x \Rightarrow$ **exit (float)**

signifie "dans le contexte où 1 et x sont des réels en virgule flottante (**float**), et '/' est une fonction à deux arguments de type flottant qui renvoye un résultat de type flottant, l'expression 1/x a un résultat de type flottant".

La sémantique statique inclut :

- des enregistrements, des unions et des types récursifs définissables par l'utilisateur,

- du sous-typage,
- des variables réinscriptibles comme dans les langages impératifs, mais en assurant que toute lecture de variable soit précédée d'une écriture et que les variables partagées ne peuvent être utilisées pour la communication entre processus,
- des portes explicitement typées.

La sémantique dynamique est basée sur des jugements tels que $\mathcal{E} \vdash E \stackrel{\alpha(N)}{\longrightarrow} E'$. Un tel jugement signifie que 'dans l'environnement $\mathcal{E}$, l'expression E se réduit (via l'action $\alpha(N)$) à E''. Pour les expressions de données, les valeurs possibles de α sont soit une exception X, soit une action de terminaison réussie δ. Par exemple, l'expression $1/2$ se termine avec la valeur 0.5 :

$$\vdash 1/2 \stackrel{\delta(0.5)}{\longrightarrow} \textbf{block}$$

et $1/0$ génère l'exception Div :

$$\vdash 1/0 \stackrel{\mathrm{Div}()}{\longrightarrow} \textbf{block}$$

La sémantique dynamique inclut :

- des comportements communiquant par des portes avec d'autres comportements,
- des comportements ou des expressions générant des exceptions qui peuvent être capturées par des gestionnaires d'exceptions,
- des comportements temps-réel.

En fait, la sémantique des expressions est donnée en considérant que les expressions constituent une sous-classe de comportements : les expressions ne peuvent que se terminer ou générer une exception, et ne peuvent ni communiquer par des portes, ni avoir des caractéristiques temporelles. Cette unification des expressions et des comportements conduit à une sémantique plus simple et plus uniforme.

3. Le langage E-LOTOS avec modules

LOTOS n'a qu'une forme limitée de modules, qui permet d'encapsuler des types de données et des opérations, mais pas de processus. De plus, ce mécanisme ne permet pas d'abstraction : chaque objet déclaré dans un module est

exporté par celui-ci. Ces déficiences rendent LOTOS difficile à utiliser. Une analyse critique des types de données LOTOS du point de vue de leur utilisation est présentée, par exemple, dans [MUN 91].

Un des objectifs de E-LOTOS était de développer un système de modules permettant d'exporter, d'importer et de réduire la visibilité des objets, ainsi que de définir des modules génériques. Les modules utilisés doivent prendre en compte les types de données et les processus de façon à ne fournir qu'un seul système incluant les déclarations de processus, de types, d'opérations, etc. Pour l'abstraction et la réutilisation de spécifications, les notions d'interface et de module générique seront très utiles, comme nous le verrons.

Par exemple, en LOTOS, un simple routeur de paquets contenant un champ de donnée et une adresse serait spécifié comme suit :

```
specification Router[in, left, right] : noexit :=
  type Natural is
    sorts nat
    ...
  endtype
  type Data is
    formalsorts data
  endtype
  type Packet is Data
    sorts
      packet, dest
    opns
      mkpacket : dest,data -> packet
      getdest : packet -> dest
      getdata : packet -> data
      L : -> dest
      R : -> dest
    eqns forall p:packet, de:dest, da:data
      ofsort packet mkpacket (getdest (p),getdata (p)) = p
      ofsort dest getdest (mkpacket (de, da)) = de
      ofsort data getdata (mkpacket (de, da)) = da
  endtype
  type NatPacket is Packet actualizedby Natural using
    natpacket for packet
    data for nat
  endtype
  behaviour Router[in, left, right]
  where
    process Router [in, left, right] : noexit :=
    in?p:natpacket;
      (       [getdest(p) = L] -> left!getdata(p);Router [in, left, right]
        []  [getdest(p) = R] -> right!getdata(p);Router [in, left, right]
      )
    endproc
endspec
```

En plus des problèmes de lisibilité déjà discutés dans la section 2, cette spécification ne permet pas aisément la réutilisation. Par exemple, il n'est pas possible de paramétrer le processus 'Router' avec un type de donnée générique, car la partie comportementale de LOTOS n'accepte que des types complètement instanciés. Donc, pour spécifier un routeur manipulant des booléens, on doit réécrire le processus 'Router' afin qu'il accepte des 'BoolPackets' plutôt que des 'NatPackets'.

Par comparaison, nous montrons comment la même spécification peut être écrite en E-LOTOS :

```
interface Data is
  type data
endint
module Destination is
  type dest is L|R endtype
endmod
generic Router(D:Data) imports Destination is
  type packet is (de => dest, da => data) endtype
  process Router[in:packet, left:data, right:data] is
    ...
  endproc
endgen
module NatRouter is
  Router (Natural renaming (types nat := data))
endmod
specification Router imports NatRouter is
  gates in:any, left:any, right:any
  behaviour
    Router [in, left, right]
endspec
```

Remarquons que :

- Des types de données, comme 'data', peuvent être déclarés dans des interfaces.
- Les modules génériques sont paramétrés par des interfaces, selon un style fonctionnel.
- Des processus peuvent être déclarés dans les modules génériques ou non ; des types de donnée génériques peuvent être utilisés dans les expressions (de données et de comportement).
- Les modules, génériques ou non, peuvent importer d'autres modules afin d'utiliser leurs définitions.
- Les modules ont une interface par défaut, qui contient toutes les déclarations du module. Cependant, une interface peut restreindre explicitement la visibilité des objets.
- Les modules génériques sont instanciés par des modules qui peuvent être mis en correspondance avec une interface (via un renommage). Par exemple, le module 'Natural' et l'interface 'Data' peuvent se correspondre

si nous associons le type 'data' à **nat**. Ainsi, le routeur générique peut être instancié par 'Natural' pour obtenir un routeur commutant des données nombres naturels.

- Une spécification peut importer des modules déjà déclarés. Le corps d'une spécification peut être une expression de comportement ou simplement une expression. Les portes et les exceptions utilisées dans le corps doivent être déclarées.

- L'espace de noms est plat, c'est-à-dire que le module de définition ne prefixe pas les noms de types, des fonctions, des processus.

- Un ensemble de modules sont prédefinis pour les types classiques et les schémas de type.

Le langage de modules comprend des déclarations de spécification, de module, d'expression de module, de module générique, d'interface, d'expression d'interface, en plus de toutes les déclarations classiques de processus, de types, etc.

Le langage des modules de E-LOTOS est basé sur des discussions antérieures [JEF 95b] et sur des propositions d'extension de LOTOS avec un système de modules [SIG 96].

3.1. *Structuration de spécifications*

Une spécification E-LOTOS est donnée par une séquence de déclarations de *modules*, de *modules génériques*, et d'*interfaces*.

Modules

Les modules sont des séquences de *déclarations* de types, de constructeurs, de processus, de fonctions, et de (valeurs) constantes. Pour les types, fonctions, processus et constantes, l'utilisateur doit fournir une implantation. La forme la plus simple de *déclarations de module* est la suivante :

module mid **is** dec **endmod**

où 'mid' est un identificateur de module et 'dec' est une déclaration (du langage de base), enrichie par des déclarations de constantes. Par exemple, un domaine n'ayant que le seul représentant 0 serait défini comme suit :

```
module OnePoint is
  type M is zero() endtype
  value 0:M is zero() endval
  function infix + (x:M, y:M):M is
    case(x,y) is
      (!zero(),!zero()) -> 0
    endcase
  endfunc
endmod
```

Contrairement à SML, qui est assez inhabituel à cet égard, il n'est pas possible d'encapsuler des modules. L'encapsulation introduit beaucoup de complexité sans avantage crucial.

On peut déclarer des processus dans des modules comme le montre l'exemple suivant :

```
module DataFlow is
  process Flow [In:PairOfInts, Out:int] is
    In(?x=>int, ?y=>int);Out!(x+y);Flow [In, Out]
  endproc
endmod
```

Dans cet exemple, étant donné que le type **int** est un type primitif, il n'est pas besoin d'importer son module de définition pour l'utiliser.

Interfaces

Intuitivement, une *interface* est le type d'un module. Alors qu'une expression de module déclare un module, une expression d'interface spécifie une classe de modules. Par exemple, le module 'DataFlow' a pour interface :

```
interface DataFlow is
  process Flow [In:pairOfInts, Out:int]
endint
```

Une interface n'est pas le type d'un module particulier, mais plutôt d'une classe entière de modules, plus précisément la classe de tous les modules qui peuvent être *mis en correspondance* avec cette interface. Par exemple, l'interface 'DataFlow' peut être le type de tout module qui dispose d'au moins un processus appelé 'Flow', ayant des portes de types **(int, int)** et **int**, et la fonctionnalité **exit(none)**.

En E-LOTOS, nous acceptons aussi que des équations soient spécifiées dans les interfaces, comme dans l'interface 'Monoid' :

```
interface Monoid is
  type M
  value 0:M
  function infix + (M, M) : M
  eqns forall x, y, z : M ->
    (0 + x) = x;
    (x + 0) = x;
    ((x + y) + z) = (x + (y + z));
  endeqns
endint
```

Les outils pourraient considérer les spécifications équationnelles comme de simples commentaires (correctement typés), de sorte que la sémantique dynamique les ignore (comme c'est le cas en Extended ML).

Modules génériques

La généricité est un outil nécessaire à la construction des spécifications et à leur réutilisation. Les modules génériques permettent de construire des bibliothèques standard de composants. La déclaration la plus simple d'un tel module a la forme suivante :

generic gid (mid:intid) **is** dec **endgen**

où 'gid' est un identificateur de module générique, 'mid' est un identificateur de module formel qui peut être mis en correspondance avec (typé par) l'interface 'intid', et 'dec' sont des déclarations du langage de base.

Nous remarquons donc que, en plus de la spécification des objets exportés par un module, les interfaces sont aussi utilisées pour spécifier les paramètres d'un module générique. Par exemple, un module définissant une liste générique peut être implanté sous forme de listes monomorphes comme suit :

```
interface List imports Monoid is
  type E
  function inj(x:E):M
endint
interface EqType is
  type E
endint
```

```
generic GenericList (Eq:EqType):List is
  type M is nil() | cons(E, M) endtype
  value 0 is nil() endval
  function infix + (s1:M, s2:M):M is
    case s1 is
        nil() → s2
      | cons(?x, ?xs) → cons(x, xs+s2)
    endcase
  endfunc
  function inj(x:E):M is cons(x, nil()) endfunc
endgen
```

Ce module peut être utilisé comme suit :

```
module ListNat:[List renaming (types E := nat)] is
  GenericList(Natural renaming (types nat := E))
endmod
```

La présence des opérations de renommage respectivement dans l'interface 'List' et dans le module 'Natural' est nécessaire afin que, dans le premier cas, l'interface de 'ListNat' contienne le type **nat** et non 'E', et dans le second cas, que le module 'Natural' soit importé en mettant en correspondance le type primitif **nat** avec le type 'E' du module générique 'GenericList'.

Notons encore que les modules génériques ne peuvent pas être paramétrés par des modules génériques.

3.2. *Abstraction*

L'abstraction peut être obtenue par le biais de déclarations de types abstraits de données (ADT) dans les interfaces. Un type abstrait de données est un type dont l'implementation n'est pas spécifiée. Par exemple, les types 'Element' et 'Set' ci-dessus :

```
interface Set is
  type Element
  type Set
  value empty:Set
  function insert(x:Element, s:Set):Set
  function delete(x:Element, s:Set):Set
  function member(x:Element, s:Set):bool
endint
```

Le type abstrait peut être réalisé (ou concrétisé) par de multiples types de données. Par exemple, plusieurs réalisations du type abstrait 'Set' peuvent être données : par des listes, des arbres binaires, ...

Un aspect important associé aux types abstraits de donnée est la notion d'égalité pour les valeurs d'un type. En E-LOTOS, vu la synchronisation, l'égalité est requise sur tout type que l'on souhaite utiliser pour typer les valeurs communiquées par des portes. Jusqu'ici, tous les types étaient munis d'une notion d'égalité, mais les modules, par l'abstraction qu'ils apportent, peuvent faire en sorte que la représentation interne d'un type de donnée est inconnue, donc aussi la notion d'égalité entre valeurs de ce type. Pour éviter cela, le langage impose qu'une fonction d'égalité soit définie pour chaque type abstrait de donnée utilisé dans les communications.

Un autre moyen de réaliser l'abstraction est le camouflage (hiding). Par le biais d'une interface appropriée, il est possible de restreindre la visibilité des objets d'un module au sous-ensemble souhaité. Il est en outre possible de définir plusieurs interfaces différentes pour un même module. On peut les considérer comme des vues différentes d'un module. Le module doit toutefois pouvoir être mis en correspondance avec ces interfaces. Considérons la spécification suivante :

```
interface VIEW1 is
  type S
  value x:S
  value y:S
endint
interface VIEW2 is
  type S
  type pairS is (x=>S, y=>S) endtype
endint
module A is
  type S is ACK()|REQ() endtype
  value x : S is ACK() endval
  value y : S is REQ() endval
  type pairS is (x=>S, y=>S) endtype
endmod
module A1:VIEW1 is A endmod
module A2:VIEW2 is A endmod
```

Il résulte de l'interface 'VIEW1' que seuls les composants spécifiés dans 'VIEW1' seront accessibles aux utilisateurs du module 'A1'. En particulier, les valeurs 'x' et 'y' ne seront pas accessibles via 'A2'.

3.3. *Composition de modules et d'interfaces*

Les modules et interfaces peuvent être composés de plusieurs façons :

- Par importation d'interfaces dans des interfaces en utilisant une clause "**imports** $iexp_1, \cdots, iexp_n$" où '$iexp_j$' sont des expressions (identificateur, renommage) d'interfaces.
- Par importation de modules dans des modules ou des modules génériques en utilisant "**imports** $mexp_1, \cdots, mexp_n$", où '$mexp_j$' sont des expressions (identificateur, renommage) de modules.
- Par renommage d'interfaces ou de modules, dont la forme primitive est :

  module mid′ **is**
  mid **renaming (types** S := S', ... **opns** C := C', ...)
  endmod

  Le type S et l'opérations C sont declarés dans le module mid ; les types S' et l'opération C' renomment respectivement S et C dans le module mid′.
- Par instanciation de modules génériques, dont la forme primitive est :

  module mid **is** gid(mid′) **endmod**

Importation d'interface

Des interfaces peuvent importer d'autres interfaces. Par exemple, l'interface pour préordres étend celle des ordres partiels avec une équation :

```
interface PreOrder is
  type T
  function infix <=(x:T, y:T):bool
  eqns forall x, y, z:T
    x<=x;
    (x<=y andalso y<=z)=>x<=y;
  endeqns
endint
interface PartialOrder imports PreOrder is
  eqns forall x, y:T
    (x<=y andalso y<=x)=>x=y;
  endeqns
endint
```

Remarquons que le type **bool** et les opérations sur ce type (comme '=> ' et **andalso**) sont prédéfinis.

Importation de modules

Des modules peuvent importer d'autres modules : la restriction des nombres naturels (que nous supposons être définis par le module 'Natural') à une structure de monoïde (définie par l'interface 'Monoid') serait réalisée par :

```
module NatMonoid:Monoid imports Natural is
  type M is nat endtype
endmod
```

Dans le cas d'une importation multiple, les définitions fournies par chaque importation doivent être compatibles. Donc, si plusieurs modules fournissent des définitions ayant le même nom, ces définitions doivent être identiques.

Instanciation

L'instantiation des modules génériques permet de réutiliser les spécifications. Par exemple, une liste générique (Cf. 'List' défini antérieurement) peut être instanciée plusieurs fois :

```
module ListNat:[List renaming (types E := nat)] is
  GenericList(Natural renaming (types nat := E))
endmod
module ListBool:[List renaming (types E := bool)] is
  GenericList(Boolean renaming (types bool := E))
endmod
```

Le résultat de la première instanciation est un module ayant pour type 'E' le type **nat**, et pour type 'M' une liste de nombres naturels. L'interface de ce module est l'interface 'List' où le type 'E' est remplacé (renommé) par **nat**. De même pour la deuxième instanciation. L'importation des deux modules 'ListNat' et 'ListBool' générerait une collision de noms pour 'M', '+' et 'inj'. Pour éviter cela, un renommage peut être appliqué, comme décrit au paragraphe suivant.

Puisque les listes, les tableaux et les ensembles sont fréquemment utilisés dans les spécifications, une syntaxe enrichie a été conçue pour eux. (Cf. 3.6).

Renommage

Le renommage est utilisé pour donner un nom unique aux objets en cas de collision de noms. La solution proposée est compatible avec le renommage des

types ACT ONE de LOTOS. Par exemple, l'obtention d'un module relatif à des listes d'entiers à partir du type 'ListNat' se ferait comme suit :

```
module ListNat is
  GenericList(Natural renaming (types nat := E))
    renaming (types ListNat := List)
endmod
```

3.4. *Relation avec l'environnement extérieur*

Des déclarations externes sont autorisées dans les modules pour permettre l'interfaçage avec d'autres langages de spécification ou de programmation. Par exemple, on pourrait donner une implantation externe de l'interface 'Monoid' en déclarant :

```
module ExtMonoid:Monoid is external endmod
```

Tout objet déclaré **external** n'a toutefois aucune sémantique dynamique formelle.

3.5. *Compatibilité avec ACT ONE*

Le système de modules d'E-LOTOS peut inclure des spécifications algébriques dans les interfaces. Par exemple, nous pouvons comparer la spécification LOTOS suivante

```
type Monoid is
  sorts M
  opns
    0: -> M
    _ + _:M,M -> M
  eqns forall x,y,z:M
    ofsort M
      x+0=x;
      0+x=x;
      (x+y)+z=x+(y+z)
endtype
```

à la déclaration E-LOTOS suivante :

```
interface Monoid is
  type M
  value 0:M
  function infix +(x:M,y:M):M
  eqns forall x,y,z:M
    (x+0)=x;
    (0+x)=x;
    ((x+y)+z)=(x+(y+z));
  endeqns
endint
module Monoid:Monoid is external endmod
```

Il existe une forte ressemblance entre ces deux spécifications. Mais nous noterons aussi quelques différences :

- LOTOS (plus exactement ACT ONE) permettait la surcharge des noms pour autant que le type de toute expression puisse être déterminé statiquement.
- En E-LOTOS, le module 'Monoid' est spécifié comme étant *toute* structure satisfaisant les axiomes, et non uniquement la structure *initiale* (au sens de la sémantique initiale) [EHR 85].

3.6. *Environnement de base*

L'*environnement de base* est une collection d'interfaces et de modules (éventuellement génériques) qui sont prédéfinis, et peuvent être utilisés dans toute spécification E-LOTOS. Ils jouent le même rôle pour E-LOTOS que la bibliothèque standard pour LOTOS.

L'environnement de base d'E-LOTOS comprend les *types prédéfinis* **bool**, **nat**, **int**, **rational**, **float**, **char**, **string**, ainsi que des schémas de déclarations pour les types énumérés, les enregistrements, les ensembles et les listes.

3.7. *Aspects sémantiques*

Pour le langage de modules, la définition formelle inclut une sémantique statique et une procédure de calcul de l'ensemble de déclarations correspondant à chaque module (la *mise à plat*).

La sémantique statique comprend :

- Le typage des modules par leur ensemble de déclarations. A cet endroit, la sémantique statique des modules utilise la sémantique statique du langage de base.
- Le typage des interfaces et des déclarations équationnelles.
- La mise en correspondance entre un module et son interface de déclaration.
- Le typage du renommage des modules et des interfaces.
- L'instanciation des modules génériques.

Références

[BOL 87] T. Bolognesi et Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14, 1987.

[DAV 92] Jim Davies, Dave Jackson et Steve Schneider. Broadcast communication for real-time processes. In J. Vytopil, ed, *Proc. Formal Techniques in Real-Time and Fault-Tolerant systems*, pages 149–170. Springer-Verlag, 1992. LNCS 571.

[EHR 85] H. Ehrig et B. Mahr. Fundamentals of algebraic specification. *Bull. Euro. Assoc. Theoret. Comp. Sci.*, 6, 1985.

[GAR 96a] Hubert Garavel et Mihaela Sighireanu. French-Romanian integrated proposal for the user language of E-LOTOS. Rapport SPECTRE 96-05, VERIMAG, Grenoble, May 1996. Input document (KC3) to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Kansas City, Missouri, USA, May, 12–21, 1996.

[GAR 96b] Hubert Garavel et Mihaela Sighireanu. On the introduction of exceptions in LOTOS. In R. Gotzhein et J. Bredereke, eds, *Formal Description Techniques IX and Protocol Specification, Testing and Verification XVI*, pages 469–484. Chapman & Hall, London, 1996.

[HER 98] C. Hernalsteen. *Specification, Validation and Verification of Real-Time Systems in ET-LOTOS*. Doctoral thesis, Free University of Brussels, Belgium, 1998.

[HOA 85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ISO 89] ISO/IEC-JTC1/SC21. *LOTOS—A formal description technique based on the temporal ordering of observational behaviour*, 1989. ISO 8807.

[ISO 00] ISO/IEC-JTC1/SC7. Information technology - E-LOTOS. ISO/IEC DIS 15437, October 2000.

[JEF 95a] Alan Jeffrey. Semantics for a fragment of LOTOS with functional data and abstract datatypes. In *Revised Working Draft on Enhancements to LOTOS (v3)*, ISO/IEC JTC1/SC21/WG7 N1053, chapter Annexe A. 1995.

[JEF 95b] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur et Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21 N10108 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[JEF 96a] Alan Jeffrey. A core data and behaviour language for E-LOTOS. Input document (KC1) to the ISO/IEC JTC1/SC21/WG7/E-LOTOS meeting in Kansas City, May 1996.

[JEF 96b] Alan Jeffrey et Guy Leduc. E-LOTOS core language. Chapter 3 of [**?**], 1996.

[LÉO 97] Luc Léonard et Guy Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29(3) :271–292, 1997.

[LÉO 98a] Luc Léonard et Guy Leduc. A formal definition of time in LOTOS. *Formal Aspects of Computing*, 10 :248–266, 1998.

[LÉO 98b] Luc Léonard et Guy Leduc. A formal definition of time in LOTOS. *Formal Aspects of Computing - Electronic journal version*, 10 :28–96, 1998. http ://www.link.springer.de/link/service/journals/00165/supp/1998/8010030248.pdf.

[MIL 89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MIL 90] Robin Milner, Mads Tofte et Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MUN 91] Harold B. Munster. LOTOS specification of the MAA standard, with an evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[SIG 96] Mihaela Sighireanu et Hubert Garavel. On the definition of modular E-LOTOS. Input document (GR2) to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Grenoble, France, December 9-11, 1996, December 1996.