A Wish List for the Behaviour Part of LOTOS

Version 2.0

Hubert GARAVEL[*]
INRIA Rhône-Alpes
VERIMAG — Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE
Tel : +(33) 76 90 96 34
Fax : +(33) 76 41 36 20
E-mail : `hubert.garavel@imag.fr`

December 1995

Abstract

This document supersedes a previous AFNOR contribution dated from June 1994 and entitled "Six improvements to the process part of LOTOS". We propose nineteen changes, which affect the syntax, static semantics and/or dynamic semantics of the behaviour part of LOTOS. These changes aim at solving several problems found in LOTOS and making the behaviour part of E-LOTOS expressive, simple, symmetric with the data part of E-LOTOS, and compatible with the usual notations found in major programming languages.

Introduction

This document supersedes a previous AFNOR contribution, dated from June 1994, entitled "Six improvements to the process part of LOTOS" [Gar94b].

This paper proposes nineteen modifications of the Formal Description Technique LOTOS (ISO standard 8807 [ISO88b]). The proposed changes are listed below and will be presented in the following sections:

1. Giving a printable name to the "δ" gate
2. Turning the specification identifier into an ordinary process identifier
3. Turning the reserved keyword "**i**" into a predefined gate identifier
4. Introducing two "**case**" operators
5. Introducing an "**if**" operator

[*]This work has been supported in part by the Commission of the European Communities, under project ISC-CAN-65 "EUCALYPTUS-2: A European/Canadian LOTOS Protocol Tool Set".

6. Extending the "**let**" operator
7. Introducing a "**rename**" operator
8. Removing the "**choice**" and "**par**" operators on gate lists
9. Using a bracketed syntax
10. Introducing a "**par**" operator on finite value domains
11. Introducing "n among m" synchronization
12. Unifying the ";" and ">>" operators syntactically
13. Unifying the ";" and ">>" operators semantically
14. Introducing exceptions in the behaviour part
15. Introducing iterators in the behaviour part
16. Removing the "**where**" clause from process definitions
17. Simplifying process definitions
18. Abbreviating gate parameters lists
19. Abbreviating value parameters lists

In the present document, all these modifications are presented separately one from each other, so that each section can be read independently. If one modification relies upon another one, the dependence is explicitly stated.

1 Giving a printable name to the "δ" gate

The dynamic semantics of LOTOS makes use of a special gate, the termination gate noted "δ" in the ISO standard. Due to the "**exit**" operator, the "δ" gate is never used explicitly in LOTOS descriptions. However, when an "**exit**" is executed, a rendez-vous on the δ gate is performed. At this point, a problem arises because "δ" is not a printable name using Latin character sets. For this reason, LOTOS tools usually replace "δ" by a printable identifier: "**d**", "**delta**", "**Delta**", "**exit**", etc.

To promote inter-operability between LOTOS tools, one should agree upon a printable identifier for the "δ" gate. The "**exit**" identifier seems to be a good candidate for at least two reasons:

- It is the most intuitive solution: an "**exit**" operator in the LOTOS description leads to an "**exit**" rendez-vous in the corresponding labelled transition system.
- It prevents name clashes with user-defined gate identifiers: "**exit**" is already a reserved keyword in LOTOS: users are not allowed to declare gates with the name "**exit**".

There are two different ways to implement the proposed change in the revised LOTOS standard:

1. The first solution would consist in adding a note stating that, whenever the "δ" gate is to be read or written using a Latin character set, then the name "**exit**" has to be used for this purpose.
2. A simpler solution would be to replace all occurrences of "δ" by "**exit**" in the dynamic semantics.

Both proposed changes are fully upward compatible, in the sense that any valid LOTOS description under the existing standard would remain valid under the revised standard.

2 Turning the specification identifier into an ordinary process identifier

Each LOTOS description is given an identifier (introduced by the "**specification**" keyword). This identifier is very similar to process identifiers (introduced by the "**process**" keyword) but not completely, as [ISO88b] imposes several distinctions between specification and process identifiers:

- Specification identifiers and process identifiers belong to distinct name spaces. For a given LOTOS description, the specification identifier name space only contains a single element (the name of the description).
- There is no place in a LOTOS description where the specification identifier can be used.
- Consequently, it is not allowed to use the specification identifier in place of a process identifier. In particular, the specification identifier cannot be used in a process instantiation [ISO88b, 7.3.4.2.a]. Therefore, a LOTOS description cannot be directly recursive:

  ```
  specification S [G] : noexit behaviour
     G; S [G]   (* illegal: identifier S cannot be used here *)
  endspec
  ```

  Recursion can only be expressed by introducing an auxiliary process:

  ```
  specification S [G] : noexit behaviour
     P [G]
  where
     process P [G] : noexit :=
        G; P [G]
     endproc
  endspec
  ```

There is very little justification for these constraints regarding specification identifiers. One can only think of one reason: by preventing the specification identifier from being a process identifier, one may wish to maintain a fair balance between data part and process part (avoiding the supremacy of processes over types at the top level of a LOTOS description). However, this is not true, since static and dynamic semantics rules already consider the LOTOS specification as a special process.

We propose the following changes, in order to simplify the revised LOTOS standard:

- The specification identifier should be a process identifier.
- The specification identifier should be visible in the behaviour expression following the "**behaviour**" (or "**behavior**") keyword.

The proposed change is fully upward compatible.

3 Turning the reserved keyword "i" into a predefined gate identifier

In standard LOTOS, the identifier of the invisible gate "**i**" is a reserved keyword. Consequently, it is not possible to declare any object (type, sort, operation, variable, process, or gate) named either "**i**" or "**I**". This situation is annoying for several reasons:

- Identifiers "**i**" and "**I**" are widely used in computer programs, due to traditions inherited from common mathematical practice and early programming languages such as FORTRAN. The prohibition of these identifiers in LOTOS is confusing to most users.
- There are some situations where the "**i**" identifier would be especially appropriate, for instance when dealing with complex numbers, matrix indexes, etc. For example, the following type definition is rejected:

```
type CHARACTER is
   sorts CHAR
   opns
      A : -> CHAR    B : -> CHAR    C : -> CHAR
      D : -> CHAR    E : -> CHAR    F : -> CHAR
      G : -> CHAR    H : -> CHAR    I : -> CHAR (* defining "I" is illegal *)
      J : -> CHAR    K : -> CHAR    L : -> CHAR
      M : -> CHAR    N : -> CHAR    O : -> CHAR
      P : -> CHAR    Q : -> CHAR    R : -> CHAR
      S : -> CHAR    T : -> CHAR    U : -> CHAR
      V : -> CHAR    W : -> CHAR    X : -> CHAR
      Y : -> CHAR    Z : -> CHAR
endtype
```

This problem could be easily solved by applying the following changes to the existing LOTOS standard:

1. Terminal symbols "**i**" and "**I**" should be removed from LOTOS BNF syntax and, therefore, should loose their status of reserved keywords. Practically, the two grammar rules below should be deleted:

```
<internal-event-symbol> ::= "I" ;
<action-denotation> ::= <internal-event-symbol> ;
```

2. The revised standard should introduce one predefined gate identifier noted "**i**" (also equivalent to "**I**").
3. The use of the predefined gate "**i**" should be restricted by introducing static semantics constraints, in order to maintain compatibility with existing LOTOS.

   Currently, due to syntax rules, the use of the "**i**" gate is strictly restricted. The idea is to shift these restrictions from syntax to static semantics. The revised standard should therefore contain the following static semantics constraints:

   - The "**i**" gate cannot occur in any gate definition context (i.e. binding occurrence), meaning that it is forbidden to declare any gate of name "**i**". The following constructs are therefore prohibited:

   ```
   hide i in ...

   choice i in [...] ...

   par i in [...] ...

   process P [..., i, ...] ...
   ```

   - The "**i**" gate cannot occur in any gate definition context (i.e. place-marking occurrence), except on the left-hand side of an action-prefix operator without experiment-offer. The following constructs are therefore prohibited[1]:

```
choice ... in [..., i, ...]

par ... in [..., i, ...]

P [..., i, ...]

i !... ; ...

i ?... ; ...
```

4. Since name spaces are considered to be distinct in LOTOS, identifiers "**i**" and "**I**" would be predefined only for gates, but would remain available for types, sorts, variables, operations, and processes.

This change is fully upward compatible.

4 Introducing two "case" operators

The output document of the Ottawa meeting [JGL+95] agreed that it would be desirable to enforce a symmetry between the data part and the behaviour part of E-LOTOS. According to the recommendations of Section 5.3.7 and Section 9.2 of [JGL+95], we propose to introduce a "**case**" operator in the behaviour part.

We base our proposal on the proposal for the E-LOTOS data type language made in [SG95]. In the sequel, let's assume the following definitions:

- $B, B', B'', B_0, B_1, B_2, ...$ denote behaviour expressions of the behaviour language.
- $E, E', E'', E_0, E_1, E_2, ...$ denote value expressions of the data type language.
- $P, P', P'', P_0, P_1, P_2, ...$ denote pattern expressions of the data type language.
- $M, M', M'', M_0, M_1, M_2, ...$ denote match expressions of the data type language. Match expressions have the following syntax:

$$\begin{array}{rcl} M & ::= & E :: P \\ & | & M \ \textbf{where}\ E \\ & | & M_1 \ \textbf{and}\ M_2 \\ & | & M_1 \ \textbf{or}\ M_2 \end{array}$$

The proposal made in [SG95] contains two "**case**" operators: the general "**case**" operator and the usual "**case**" operator.

- Three new keywords have to be added: "**case**", "**otherwise**", and "**endcase**".
- For the general case operator, the following rule has to be added to the BNF grammar:

[1]Unrestricted use of the "**i**" gate would lead to subtle problems, such as action denotations of the form "**i** !v_1 ... !v_n", which are not handled in the existing dynamic semantics of LOTOS

$$
\begin{array}{lll}
B & ::= & \textbf{case} \\
 & & M_0 \rightarrow B_0 \\
 & & \ldots \\
 & & M_n \rightarrow B_n \\
 & & [\textbf{otherwise } B_{n+1}] \\
 & & \textbf{endcase}
\end{array}
$$

As stated in [JGL+95] and [SG95], the evaluation of a "**case**" operator is sequential and deterministic. The match expressions M_i are evaluated in turn:

- If it exists, the smallest i such that M_i "matches" is selected, and the corresponding B_i is executed.
- If no M_i matches and if the "**otherwise**" clause if present, then B_{n+1} is executed.
- If no M_i matches and if the "**otherwise**" clause if absent, then several semantics are possible:
    * One could say that nothing happens, i.e., behave as if an "**otherwise stop**" clause was present.
    * Another approach is to avoid this problem by making the "**otherwise**" clause mandatory.
    * However, the two above semantics are not symmetric with the data language, in which incomplete "**case**" operators are used to describe partial functions and provoke "run time" errors. One could therefore say that a missing case in the behaviour part either causes the whole behaviour to be undefined (the so-called "core dump" semantics) or raises an exception.

The "**otherwise** B_{n+1}" clause is a shorthand which can be expanded to "$\mathit{true}::\mathit{true} \rightarrow B_{n+1}$".

- For the usual case operator, the following rule has to be added to the BNF grammar:

$$
\begin{array}{lll}
B & ::= & \textbf{case } E \textbf{ in} \\
 & & P_0^0|\ldots|P_{m_0}^0 \ [\textbf{where } E_0] \rightarrow B_0 \\
 & & \ldots \\
 & & P_0^n|\ldots|P_{m_n}^n \ [\textbf{where } E_n] \rightarrow B_n \\
 & & [\textbf{otherwise } B_{n+1}] \\
 & & \textbf{endcase}
\end{array}
$$

Remark
The "**case...in**" syntax was preferred to the "**case...of**" syntax found in PASCAL in order to avoid a parsing conflict: LOTOS expressions may already contain "**of**" operators. □

The usual case operator is merely a shorthand, which can be expanded as follows to a general case operator:

```
        case
        $E::P_0^0$ or ... or $E::P_{m_0}^0$ [where $E_0$] $\rightarrow B_0$
        ...
        $E::P_0^n$ or ... or $E::P_{m_n}^n$ [where $E_n$] $\rightarrow B_n$
        [otherwise $B_{n+1}$]
        endcase
```

or, if one wants (for efficiency reasons) to evaluate expression E only once, by introducing a variable X whose type T is the type of E:

```
        case
        $E::X:T \rightarrow$
            case
            $X::P_0^0$ or ... or $X::P_{m_0}^0$ [where $E_0$] $\rightarrow B_0$
            ...
            $X::P_0^n$ or ... or $X::P_{m_n}^n$ [where $E_n$] $\rightarrow B_n$
            [otherwise $B_{n+1}$]
            endcase
        endcase
```

Remark
The guard operator "$[E] \rightarrow B$" that exists in LOTOS can be expressed as a particular form of "**case**" operator:

```
        case $E$ in
        $true \rightarrow B$
        $false \rightarrow$ stop
        endcase
```

□

Remark
Reciprocally, it is not possible to reduce "**case**" operators to a combination of guards and non-deterministic choices, because of the variable bindings resulting from pattern-matching. □

It is clear that "**case**" operators cannot be reduced to existing LOTOS operators: pattern-matching brings new expressiveness, which does not directly exists in LOTOS. For instance, the use of pattern-matching will highly simplify the frequent situation in which a protocol receives a packet and makes different actions depending upon the packet type and the value of the packet fields. Pattern-matching will allow packet type recognition and packet field extractions all at once.

Also, the "**case**" operator should improve run-time efficiency, since when the type of a packet has been recognized, it is not necessary to check for the other possible packet types.

This extension is upward compatible, except for those existing LOTOS programs that use the new reserved keywords. For those programs, a renaming of conflicting identifiers would be needed.

5 Introducing an "if" operator

Due to its process algebra origins, LOTOS uses only two primitives (guards and non-deterministic choice) to express conditionals. This imposes a specification style that is not intuitive (novice users are not familiar with "guarded commands" and usually prefer the classical "**if-then-else**" constructs), tedious to write, difficult to read (guarded commands are more verbose than their "**if-then-else**" equivalents), and error-prone.

For these reasons, we propose to extend LOTOS with an "**if**" operator[2]. The following changes should be introduced in the revised LOTOS standard:

- Five new keywords have to be added: "**if**", "**then**", "**else**", "**elsif**", and "**endif**".
- The following rule has to be added to the BNF grammar:

$$\begin{array}{lll} B & ::= & \textbf{if } E_0 \textbf{ then } B_0 \\ & & \textbf{elsif } E_1 \textbf{ then } B_1 \\ & & \ldots \\ & & \textbf{elsif } E_n \textbf{ then } B_n \\ & & [\textbf{else } B_{n+1}] \\ & & \textbf{endif} \end{array}$$

  Annex A gives a concrete syntax for the "**if**" operator and explains how it can be expanded into standard LOTOS (using combinations of guards and deterministic choices). But, assuming the existence of the general "**case**" operator, the translation scheme suggested in [SG95] is much simpler:

$$\begin{array}{l} \textbf{case} \\ E_0 :: true \rightarrow B_0 \\ E_1 :: true \rightarrow B_1 \\ \ldots \\ E_n :: true \rightarrow B_n \\ [\textbf{otherwise } B_{n+1}] \\ \textbf{endcase} \end{array}$$

  This translation is likely to be implemented more efficiently than the one given in Annex A, because the expressions E_i are evaluated only if necessary. But, if run-time errors (e.g., exceptions) are introduced in the behaviour part, then the evaluation of expressions E_i may raise a run-time error. In such case, the two translation schemes are not equivalent, and the one given above should be preferred, as it expresses the "**if**" semantics adequately.

This extension is upward compatible, except for those existing LOTOS descriptions that contain identifiers with the same spelling as the new reserved keywords. For those descriptions, renaming the conflicting identifiers would be needed.

[2]This operator has exactly the same syntax and semantics as in Ada

6 Extending the "let" operator

The output document of the Ottawa meeting [JGL+95] agreed that it would be desirable to enforce a symmetry between the data part and the behaviour part of E-LOTOS. According to the recommendations of Section 9.2 of [JGL+95], we propose to extend the expressiveness of the existing "**let**" operator to allow pattern-matching.

We base our proposal on the proposal for the E-LOTOS data type language made in [SG95].

- No new keyword must be added.
- The existing BNF grammar should be modified: the existing definition of the "**let**" operator should be replaced with the following rule:

$$B \quad ::= \quad \textbf{let } P_0 \texttt{:=} E_0, ..., P_n \texttt{:=} E_n \textbf{ in } B_0$$

  Like the "**if**" operator, the "**let**" construct is merely a shorthand for notational convenience, which can be expanded to a general "**case**" operator:

  case
  $E_0 \texttt{::} P_0$ **and** ... **and** $E_n \texttt{::} P_n \rightarrow B_0$
  endcase

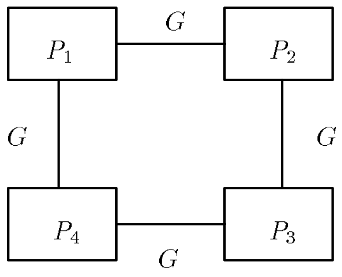
This modification is "almost" upward compatible with existing LOTOS because E-LOTOS patterns include variable declarations "$V : T$" as a particular form of patterns. To translate a LOTOS "**let**" operator into an E-LOTOS "**let**" operator, two changes have to be performed:

1. Variable lists of the form "$V_1, ..., V_n$**:**T" must be flattened, i.e., replaced with "$V_1 : T, .., V_n : T$". However, such variables are seldom used in LOTOS programs because it is not very useful to define several variables with different names and the same value.
2. The "=" symbols used in LOTOS "**let**" operators must be replaced with "`:=`" symbols.

7 Introducing a "rename" operator

We suggest to introduce in E-LOTOS a "renaming" operator which would allow to modify the gates and/or the offers of the labels of the actions performed by a given behaviour. The underlying motivations for this change are the following:

1. It is well-known that the parallel composition operators of LOTOS are not enough to express all possible forms of synchronization. This is the case, for instance, of the circular ring of four processes $P_1, ..., P_4$ depicted below:



Renaming allows to express some networks of parallel processes, which could not otherwise be obtained by simply using the parallel composition operators of LOTOS. For instance, by introducing an auxiliary gate G', which is later renamed in G, the ring can be described as follows:

```
rename G' := G in
  (
    (P1[G', G] |[G]| P2[G, G] |[G]| P3[G, G])
    |[G']|
    P4[G, G']
  )
```

2. Renaming is also needed for reusability purpose. Let's consider two LOTOS behaviours B_1 and B_2 that, for some reason, we are not allowed to modify or to duplicate, and have to take "as is" instead. This situation will occur if B_1 and B_2 belong to a library whose components are reusable LOTOS processes. Let assume that B_1 and B_2 are to be assembled in a pipeline structure: they are put in parallel and synchronized together on a common gate G. Let's assume that B_1 sends messages on G which are received by B_2. A problem occurs if the messages sent by B_1 are not in the same format as those expected by B_2. By example:

   - B_1 could send messages of the form "G !E" and B_2 expect messages of the form "G !E !$f(E)$", where f is a checksum function, for instance — or vice-versa.
   - B_1 could send messages of the form "G !E" and B_2 expect messages of the form "G !*header* !E !*trailer*" (according the layering principle of OSI) — or vice-versa.
   - B_1 could send messages of the form "G !E" and B_2 expect messages of the form "G !$f($*header*$, E,$ *trailer*$)$", where f is a packet-making function, for instance — or vice-versa.

   Using standard LOTOS, the single solution to this problem is to introduce an "interface" behaviour between B_1 and B_2, which is more or less a one-slot buffer performing data conversion. Although tractable, this approach reduces the efficiency of the implementation (an auxiliary process must be added in parallel, and auxiliary synchronizations and communications have to be performed at run-time) and increases the number of states in the global system, thus contributing to state explosion.

   On the opposite, the renaming operator described below solves the problem, avoiding the introduction of an "interface" process. It also improves reusability, by allowing to define multiple "views" of an existing software component (for instance, process components such as buffers, queues, "chaos" processes, etc., could be reused easily, which is not the case currently in LOTOS).

3. Renaming can be useful for verification purpose, especially when using bisimulation relations. In such case, one often wants to rename all visible actions of "no interest" for the property to be verified into some special action A. Therefore, the renaming can be non-injective. Hiding is not sufficient for this goal if this action A must remain visible to be distinguished from the internal action "**i**".

4. The dynamic semantics of standard LOTOS already has a "relabelling" operator. However, this operator has two limitations:

   - It allows to change the gate names in labels, but not the offers.

- It is "hidden" in the dynamic semantics and cannot be used directly, nor simply, in LOTOS descriptions. To use this relabelling operator, one must define a process and invoke this process with different gate parameters. If $G, G', G'', G_0, G_1, G_2, ...$ are gates, the specifier cannot simply write:

$$\textbf{rename } G_0 \text{ := } G'_0, ..., G_n \text{ := } G'_n \textbf{ in } B_0$$

but has to write instead:

$$\begin{array}{l} P[G'_0, ..., G'_n, \mathcal{G}](\mathcal{E}) \\ \textbf{where} \\ \textbf{process } P[G_0, ..., G_n, \mathcal{G}](\mathcal{V}) : \mathcal{F} \text{ :=} \\ \quad B_0 \\ \textbf{endproc} \end{array}$$

where P is a (new) process name, $\mathcal{G}$ denotes the list of the visible gates in B_0 different from $G_0, ..., G_n$, $\mathcal{V}$ denotes the list of variables used in B_0, $\mathcal{E}$ denote the list of the values to be assigned to the variables of $\mathcal{V}$, and $\mathcal{F}$ denotes the functionality of process P.

Making the renaming operator explicit will make the "substitution principle" valid for LOTOS: this principle states that every process instantiation can be replaced with the process definition, modulo appropriate renamings. In the case of LOTOS, the following property will hold:

$$\left(\begin{array}{l} P[G'_1, ..., G'_m](E_1, ..., E_n) \\ \textbf{where} \\ \textbf{process } P[G_1, ..., G_m](V_1 : T_1, ..., V_n : T_n) ... \\ \quad B \\ \textbf{endproc} \end{array} \right) = \left(\begin{array}{l} \textbf{rename } G_1 \text{ := } G'_1, ..., G_m \text{ := } G'_m \textbf{ in} \\ \textbf{let } V_1 : T_1 = E_1, ..., V_n : T_n = E_n \textbf{ in} \\ B \\ \textbf{where} \\ \textbf{process } P[G_1, ..., G_m](V_1 : T_1, ..., V_n : T_n) ... \\ \quad B \\ \textbf{endproc} \end{array} \right)$$

5. Other process algebras, such as ACP or CCS already have a renaming operator, which is very general: any total function ρ mapping labels to labels can be used. The corresponding dynamic semantics is simple: when a label L is observed, it is replaced with $\rho(L)$.

   Therefore, the result of the function may depend upon the value of the offers occurring in the labels. For instance, one can define $\rho(G \ \ !\mathit{true}) = A$ and $\rho(G \ \ !\mathit{false}) = B$.

   However, we believe that such an unrestricted approach is not appropriate for LOTOS, as it would be difficult to integrate in compilers using intermediate models based on Petri Nets or Extended Finite State Machines.

   We propose to restrict renaming to functions ρ whose effects are *statically decidable*. More precisely, if L is a label containing free variables, the value of $\rho(L)$ should be computable at compile-time. Therefore, the definition of ρ should only rely on statically computable informations: the name of the gate, the number of offers and the type of the offers.

We introduce the following definitions:

- One new keyword has to be added: "**rename**".
- The following rules have to be added to the BNF grammar:

$$B \quad ::= \quad \textbf{rename } R_0, ..., R_n \ [R_{n+1}] \textbf{ in } B_0$$

where $R_0, ..., R_n$ $(0 \leq i \geq n)$ are "renaming clauses" whose syntax is:

$$\begin{array}{lll} R_i & ::= & G_i \texttt{ := } G_i' \\ & | & G_i \texttt{ (} V_i^1 : T_i^1, ..., V_i^{p_i} : T_i^{p_i} \texttt{) := } G_i' \texttt{ (} E_i^1, ..., E_i^{q_i} \texttt{)} \end{array}$$

and where R_{n+1} has the following syntax:

$$\begin{array}{lll} R_{n+1} & ::= & \texttt{... := } G_{n+1}' \\ & | & \texttt{... := } G_{n+1}' \texttt{(} E_{n+1}^1, ..., E_{n+1}^{q_{n+1}} \texttt{)} \end{array}$$

- There are additional static semantic constraints.

  For each renaming clause R_i, the list of types $T_i^1, ..., T_i^{p_j}$ should be compatible with the types of the experiment offers permitted for gate G_i (assuming that a gate typing mechanism is introduced in E-LOTOS).

  For each renaming clause R_i having a "(...)" clause (i.e., the syntax of R_i is given by the second rule of the above BNF grammar), the variables V_i^j $(0 \leq j \leq p_i)$ are visible in the expressions E_i^k $(0 \leq k \leq q_i)$.

  The renaming clauses $R_0, ..., R_n$ should be pairwise disjoint, in order not to overlap. If gates are not overloaded, this can be simply expressed as:

$$(\forall i, j \in \{0, ..., n\}) \ i \neq j \Longrightarrow G_i \neq G_j$$

  If gates can be overloaded, this constraint is more complex:

$$\begin{array}{l} (\forall i, j \in \{0, ..., n\}) \ i \neq j \Longrightarrow \\ (G_i \neq G_j) \ \vee \ ((\textit{both } R_i \textit{ and } R_j \textit{ have a } (...) \textit{ clause}) \ \wedge \ (p_i \neq p_j) \ \vee \ (\exists k \mid T_i^k \neq T_j^k)) \end{array}$$

- Informally, the dynamic semantics of the renaming operator is defined as follows:

  - Renamings R_i of the form "G_i := G_i'" behave as the existing relabelling operator of standard LOTOS. They apply to all labels whose gate is equal to G_i. They modify these labels by replacing G_i with G_i'.
  - Renaming R_i of the form "G_i ($V_i^1 : T_i^1, ..., V_i^{p_i} : T_i^{p_i}$) := G_i' ($E_i^1, ..., E_i^{q_i}$)" apply to all labels whose gate is equal to G_i and whose experiment offers have types $T_i^i, ..., T_i^{p_i}$ respectively. If these labels have the form $G_i \ v_1, ..., v_{p_i}$, they are modified as follows: G_i is replaced with G_i' and the list of offers is replaced with the list of value expressions $E_i^1, ..., E_i^{q_i}$ in which all variables V_i^j are replaced with v_i^j respectively.
  - If present, R_{n+1} plays the role of a default renaming, which applies to all labels that do not match some R_i $(0 \leq i \leq n)$.

Remark
The two renaming clauses "G := G'" and "$G'()$:= $G'()$" are not identical: the former applies to all labels having gate G, the latter to all labels having gate G and no experiment offers. □

- Formally, the dynamic semantics of the renaming operator is defined by the following rule:

$$\frac{B \xrightarrow{L} B'}{(\textbf{rename } R_0, ..., R_n \ [R_{n+1}] \textbf{ in } B) \xrightarrow{\rho[R_0,...,R_n,R_{n+1}](L)} (\textbf{rename } R_0, ..., R_n \ [R_{n+1}] \textbf{ in } B')}$$

where the renaming function $\rho[R_0, ..., R_n, R_{n+1}]$ is defined as follows:

```
$\rho[R_0, ..., R_n, R_{n+1}](G\ v_1, ..., v_l) =_{def}$
  $\textbf{if}\ \exists i \in \{0, ..., n\} \mid (G_i = G)\ \wedge\ (R_i\ \textit{has no}\ (...)\ \textit{clause})\ \textbf{then}$
    $G'_i\ v_1, ..., v_l$
  $\textbf{else\_if}\ \exists i \in \{0, ..., n\} \mid (G_i = G)\ \wedge\ (R_i\ \textit{has a}\ (...)\ \textit{clause}) \wedge$
        $(p_i = l)\ \wedge\ (\forall k \in \{1, ..., l\})\ (\textit{type}(v_k) = T_k)\ \textbf{then}$
    $G'_i\ E_i^1[V_1 := v_1, ..., V_l := v_l], ..., E_i^{q_i}[V_1 := v_1, ..., V_l := v_l]$
  $\textbf{else\_if}\ (R_{n+1}\ \textit{exists and has no}\ (...)\ \textit{clause})\ \textbf{then}$
    $G'_{n+1} v_1, ..., v_l$
  $\textbf{else\_if}\ (R_{n+1}\ \textit{exists and has a}\ (...)\ \textit{clause})\ \textbf{then}$
    $G'_{n+1} E_i^1, ..., E_i^{q_i}$
  $\textbf{else}$
    $G\ v_1, ..., v_l$
```

We give some use examples of the renaming operator:

1. simple gate renaming: **rename** G_0 `:=` $G'_0, ..., G_n$ `:=` G'_n `...` `:=` G'_{n+1} **in** ...
2. deletion of offers: **rename** $G\ (V_1 : T_1, V_2 : T_2, V_3 : T_3)$ `:=` $G(V_2)$ **in** ...
3. addition of offers: **rename** $G\ (V : T)$ `:=` $G\ (\textit{header}, V, \textit{trailer})$ **in** ...
4. duplication of offers: **rename** $G\ (V : T)$ `:=` $G\ (V, V)$ **in** ...
5. alteration of offers: **rename** $G\ (V : T)$ `:=` $G\ (0)$ **in** ...
6. modification of offers: **rename** $G\ (V : T)$ `:=` $G\ (F(V))$ **in** ...
7. permutation of offers: **rename** $G\ (V_1 : T_1, V_2 : T_2)$ `:=` $G\ (V_2, V_1)$ **in** ...
8. combination of offers: **rename** $G\ (V_1 : T_1, V_2 : T_2)$ `:=` $G\ (F_1(V_1, V_2), F_2(V_1, V_2))$ **in** ...

Remark
The semantics of the "**rename**" operator implies that:

$$\textbf{rename } G \texttt{ := } G' \textbf{ in } (G\ ;\ \textbf{stop}\ |||\ G'\ ;\ \textbf{stop}) = \textbf{stop}$$

because renaming is applied after trying to synchronize G and G', which fails, because both gates do not have the same name. This is the *post-renaming* semantics, which is also that of the relabelling operator of LOTOS.

An alternative definition, the *pre-renaming* semantics, could be used instead, which applies renaming before synchronization; this semantics is based on textual substitution of the renamed gates in the behaviour expression. Using this semantics, one would have:

$$\textbf{rename } G \texttt{ := } G' \textbf{ in } (G\ ;\ \textbf{stop}\ |||\ G'\ ;\ \textbf{stop}) = G\ ;\ \textbf{stop}$$

It is worth noticing that many compiler writers translating LOTOS into Extended Finite State Machines or Extended Petri Nets (e.g. Guether Karjoth and Carl Binding with the LOEWE tool, Eric Dubuis with the COLOS tools, and the author of this article with the CÆSAR tool) have chosen to deviate from LOTOS by implementing pre-renaming instead of post-renaming. The reasons for these deviations should be considered for the design of E-LOTOS.

Also, in the case where the relabelling function is injective (which often occurs in practice), both semantics are identical. □

This proposal for introducing renaming in E-LOTOS remains to be integrated with other proposals, such as gate typing, communication pattern-matching, time, etc.

8 Removing the "choice" and "par" operators on gate lists

We propose to remove the "**choice**" operators and "**par**" operators on gate lists, which currently exist in standard LOTOS[3]. Our motivations are the following:

- We notice that these operators are not used in practice. For instance, they are not used in the formal descriptions in LOTOS of the transport protocol, nor in the CCR service and protocol, nor in the description of the OSI-TP protocol.
- These operators are merely shorthands which do not bring expressiveness. There could be easily replaced using the renaming operator for E-LOTOS (see Section 7 above). For instance:

  $$\textbf{choice}\ G\ \textbf{in}\ \texttt{[}G_0, ..., G_n\texttt{]}\ \texttt{[]}\ B$$

  could be expressed as:

  $$\begin{array}{l}\textbf{rename}\ G := G_0\ \textbf{in}\ B\\ \texttt{[]}\\ \dots\\ \texttt{[]}\\ \textbf{rename}\ G := G_n\ \textbf{in}\ B\end{array}$$

  Remark
  A process definition could also be used, in order to avoid the duplication of B several times. □

Therefore, unless a convincing example of the practical usefulness of these two operators can be found, we propose to remove them from E-LOTOS, in order to keep the language as simple as possible.

9 Using a bracketed syntax

As LOTOS behaviour expressions are algebraic terms with nullary, unary, and binary operators, parsing ambiguities naturally arise. They are solved in two ways: by the definition of LOTOS syntax, which introduces priorities between behaviour operators, and by the specifiers, who can use parentheses to enclose behaviour expressions appropriately.

[3] We do not propose the removal of the "**choice**" operator on value domains, because this operator is used very often, for instance in the formal descriptions in LOTOS of the transport protocol and OSI-TP protocol.

However, this scheme proves to be difficult to many users. For instance, the following behaviour expression:

$$B_1 \texttt{ >> } B_2 \texttt{ [] } B_3$$

is not parsed as:

$$(B_1 \texttt{ >> } B_2) \texttt{ [] } B_3$$

but:

$$B_1 \texttt{ >> } (B_2 \texttt{ [] } B_3)$$

Experienced users solve the problem by putting lots of parentheses, which makes the specification safer for them, but difficult to read by someone else. A similar problem also occurs when several parallel operators are mixed in the same behaviour expression.

We could imagine that the BNF grammar could force the user to put parentheses where a behaviour expression is ambiguous. It seems that LOTOS syntax tries to do so, at least for the unary operators such as "**choice**", "**par**", "**let**", etc. But this results in strange syntactic constraints, which do not avoid ambiguities.

In his thesis [Bri88], Ed Brinksma addresses this problem and suggest an elegant solution, which is worth to be considered for the design of E-LOTOS. In his approach, a special syntax is introduced to "bracket" the binary operators (";", "[]", "||", "[>"). We now consider in turn the various bracketed operators proposed by Brinksma.

- The simplest one concerns the *disable* operator. It is noted:

  dis
    B_1
  `[>`
    B_2
  enddis

  and is equivalent to (B_1 `[>` B_2).

- Another bracketed operator concerns the non-deterministic choice. It is noted:

  sel
    B_0
  `[]`
    ...
  `[]`
    B_n
  endsel

  and is equivalent to (B_0 `[]` ... `[]` B_n). Note: using "**alt**...**endalt**" rather than "**sel**...**endsel**" would give a flavour of OCCAM.

- There are several forms of bracketed syntaxes for parallel composition. The first form is noted:

```
par sync all
   B_0
||
   ...
||
   B_n
endpar
```

and is equivalent to (B_0 || ... || B_n). The second form is noted:

```
par [sync none]
   B_0
||
   ...
||
   B_n
endpar
```

and is equivalent to (B_0 ||| ... ||| B_n). The third form is noted:

```
par sync G_0, ..., G_m in
   B_0
||
   ...
||
   B_n
endpar
```

and is equivalent to (B_0 |[$G_0, ..., G_m$]| ... |[$G_0, ..., G_m$]| B_n). There is also a fourth form ("**sync common**") which should be left for further study, since it is highly context dependent.

- There is also a bracketed syntax for sequential composition. It is noted:

```
seq
   A_0
;
   ...
;
   A_n
;
   B
endseq
```

and is equivalent to (A_0;...;A_n;B) where $A_0, ..., A_n$ are action denotations. However, as Brinksma also suggests to unify both operators ";" and ">>" (this will be discussed in Section 12 below), some A_i can also be replaced with behaviour expressions and "**init**" clauses.

- The syntax of the unary operator "**choice**" is changed into a bracketed syntax:

  sel for *iteration_over_gate_list_or_value_domain* []
     B
  endsel

- Similarly, the syntax of the unary operator "**par**" is changed into a bracketed syntax:

  par for *iteration_over_gate_list_or_value_domain* **sync** ...
     B
  endpar

- The unary operators "**let**" and "**hide**" are kept unchanged: no keywords "**endlet**" and "**endhide**" are added.

Note: It is not clear whether simple parentheses (...) are still allowed in LOTCAL, or if only the bracketed constructs are available.

Note: In the above list of operators, the guard operator is not mentioned. In LOTCAL, guards have a special status determined by the BNF syntax. They can be used only in arguments of "**sel**", "**dis**", "**par**", etc. For E-LOTOS, it would be probably better to keep guards as "normal" (i.e., first-class citizen) operators.

Note: most of the new operators proposed for E-LOTOS(e.g., "**case**" and "**if**") already have a bracketed syntax.

10 Introducing a "par" operator on finite value domains

We propose to introduce a "**par**" operator on value domains, similar to the "**choice**" operator on value domains. This operator would be very useful to launch a set of processes in parallel, e.g.:

$$\textbf{par } V_1{:}1..3, V_2{:}\mathit{bool} \;\;|||\;\; P[G_1, G_2](V_1, V_2)$$

would be equivalent to:

$$\begin{array}{l} P[G_1, G_2](1, \mathit{false}) \;\;|||\;\; P[G_1, G_2](1, \mathit{true}) \\ ||| \\ P[G_1, G_2](2, \mathit{false}) \;\;|||\;\; P[G_1, G_2](2, \mathit{true}) \\ ||| \\ P[G_1, G_2](3, \mathit{false}) \;\;|||\;\; P[G_1, G_2](3, \mathit{true}) \end{array}$$

However, this "**par**" operator is only well-defined if one iterates on finite value domains. Should value domains be infinite, this would lead to an unbounded number of rules in the definition of the dynamic semantics.

If data types are defined using abstract data types without constructors, as it is the case in LOTOS with ACTONE types, it is not possible to determine statically whether the domain (data carrier) of type (sort) is finite or not. This was the reason why the "**par**" operator on value domains was not introduced in LOTOS.

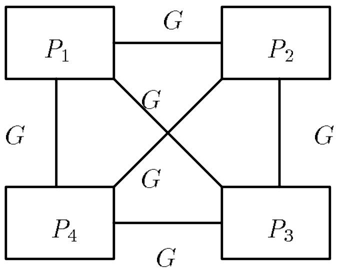
However, Ed Brinksma decided to introduce a "**par**" operator on value domains in the LOTCAL language [Bri88]: the specifier has to ensure that the value domains are finite, otherwise the meaning of the LOTCAL specification is undefined. Of course, this constraint cannot be checked statically.

But in the case of E-LOTOS, if data types are defined constructively, as recommended in the output document of the Ottawa meeting [JGL+95], it is possible to decide statically whether a type is finite or not. The algorithm is based on the following statement: *a type is finite if it is not recursive and if all the types of all the arguments of all its constructors are themselves finite.* By associating a boolean variable "*is_finite*(T)" to each type T, one obtains a system of boolean equations which can be solved iteratively. For externally-defined types, the user would have to specify whether their domain is finite (the default choice) or infinite.

11 Introducing "n among m" synchronization

Let's consider a set of m concurrent processes $P_1, ..., P_m$. We want to specify a synchronization scheme in which n ($n \leq m$) of these processes have to synchronize on a given gate G. For $n = 2$, this means that any process P_i can synchronize and communicate with any other processes P_j ($i \neq j$) using binary rendez-vous.

In standard LOTOS, specifying "n among m" synchronization is not always easy, nor even possible. Just consider, for instance, the following network of communicating processes:



However, we believe that "n among m" synchronization can be useful practically, especially for $n = 2$. We give two examples:

- In ODP systems, any object can potentially interact with any other object using binary rendez-vous.

- "Problem solving" descriptions can be obtained by putting in parallel many components, each component computing a part of the global solution and being potentially allowed to communicate with any other components. An example of such constraint-oriented descriptions is the "eight queens" problem described in FP2 by Philippe Schnoebelen. In LOTOS, it is difficult to describe a chessboard, each square of which is a parallel process being able to synchronize with adjacent squares.

In both cases, communication between processes can be restricted by using experiment offers, according to the value matching mechanism of LOTOS. A simple solution to the "eight queens" example would consist in connecting all pair of squares, then restricting synchronization to pairs of adjacent squares using appropriate offers and selection predicates.

Milner's CCS allows "2 among m" communication, but does not allow multiway rendez-vous.

We propose to combine both approaches by generalizing LOTOS parallel composition operators. Obviously, it does not seem possible to introduce the concept of "n among m" synchronization using the binary operators "||", "|||", and "|[...]|". We therefore use as a starting point the bracketed syntax proposed by Ed Brinksma for the general "**par**" operator (see Section 9 above):

```
par sync G_0, ..., G_p in
   B_1
||
   ...
||
   B_m
endpar
```

where $m \geq 1$. We propose to extend this syntax as follows:

```
par sync G_0 # n_0, ..., G_p # n_p in
   B_1
||
   ...
||
   B_m
endpar
```

where $n_0, ..., n_p$ are integer numbers called *degrees*, whose values must be computable statically (i.e., at compile-time), such that $(0 \leq n_0 \leq m) \wedge ... \wedge (0 \leq n_p \leq m)$.

Informally, the semantics of this operator is the following. The behaviour expressions $B_1, ..., B_m$ execute concurrently. The actions whose gate G does not belong to $\{G_0, ..., G_p\}$ are *asynchronous*: they can be performed by some B_i without synchronization from the other processes $B_j (j \neq i)$. The actions whose gate G is equal to some G_k are *synchronous*: they must be performed simultaneously by n_k behaviour expressions among $B_1, ..., B_m$. As in LOTOS, the termination is synchronous: all B_i's synchronize on the "δ" gate.

Remark
When $(n_0 = m) \wedge ... \wedge (n_p = m)$, the original "**par**" operator of Brinskma is obtained as a particular case. □

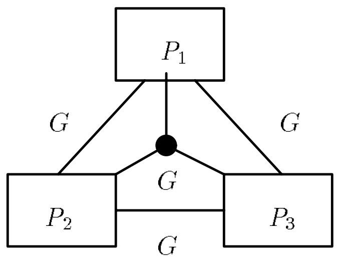
Remark
Similarly, the general parallel composition operator of LOTOS can be obtained as a shorthand:

$$B_1 \;|[G_0, ..., G_p]|\; B_2 =_{def} \left(\begin{array}{l} \textbf{par sync } G_0 \texttt{ \# } 2, ..., G_p \texttt{ \# } 2 \textbf{ in} \\ \quad B_1 \\ || \\ \quad B_2 \\ \textbf{endpar} \end{array} \right)$$

□

Remark
If some n_i is equal to 1, it means that gate G_i is asynchronous. It is therefore equivalent to have a gate not mentioned in $G_0, ..., G_n$ or to have it in the list with degree 1. □

Remark
We do not require that $G_0, ..., G_p$ are pairwise different gate identifiers. This allows different degrees of synchronization on the same gate. For instance, the following network:



can be obtained as follows:

```
par sync G # 2, G # 3 in
   B1
||
   B2
||
   B3
endpar
```

In particular, when there are only two processes, such feature can be useful to simulate Ccs parallel composition, in which parallel processes can either synchronize or evolve independently. □

Given a list "G_0 # $n_0, ..., G_p$ # n_p", we define the predicate "*has_degree*(G, d)" which is equal to *true* iff gate G has degree d. This predicate is defined as follows:

$$\mathit{has\_degree}(G,d) =_{def} \begin{cases} \textbf{if } G = \delta \textbf{ then } d = m \\ \textbf{else\_if } (\exists i) \in \{0,...,p\} \mid G = G_i \textbf{ then } d \in \{n_i | G = G_i\} \\ \textbf{else } d = 1 \end{cases}$$

We can now define the semantics of the proposed parallel composition operator with a single rule (instead of three rules in the dynamic semantics of standard Lotos):

$$\frac{\begin{array}{c}(\exists L)\ (\exists B'_1,...,B'_m)\ (\exists I \subseteq \{0,...,m\})\ \mathit{has\_degree}(\mathit{gate}(L), \mathit{card}(I)) \wedge \\ (\forall i \in I)\ (B_i \xrightarrow{L} B'_i) \wedge (\forall i \in \{0,...,m\} - I)\ (B'_i = B_i)\end{array}}{\left(\begin{array}{l}\textbf{par sync } G_0 \texttt{ \# } n_0, ..., G_p \texttt{ \# } n_p \textbf{ in} \\ \quad B_1 \\ \texttt{||} \\ \quad ... \\ \texttt{||} \\ \quad B_m \\ \textbf{endpar}\end{array}\right) \xrightarrow{L} \left(\begin{array}{l}\textbf{par sync } G_0 \texttt{ \# } n_0, ..., G_p \texttt{ \# } n_p \textbf{ in} \\ \quad B'_1 \\ \texttt{||} \\ \quad ... \\ \texttt{||} \\ \quad B'_m \\ \textbf{endpar}\end{array}\right)}$$

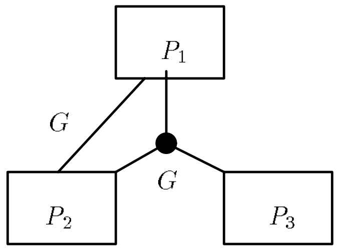
Remark
If we allow the case where, for some G_i, the corresponding degree n_i is equal to 0, the above rule will prevent $B_1, ..., B_m$ from executing any action with gate G_i. The same result could also be obtained using standard Lotos operators: given a behaviour B, if one wants to forbid actions with gate G_i, it is sufficient to write:

$$B\ \texttt{|[}G_i\texttt{]|}\ \textbf{exit}$$

□

Remark
The proposed "n among m" synchronization should be adapted to the "**par**" operator on value domains described in Section 10. □

Remark
We could also adopt another synchronization scheme, in which we associate to each gate G_i, not its degree, but the set of indexes of the B_i's which have to synchronize on this gate. This would allow to describe synchronization schemes such as this one:



which could be expressed as:

par sync G $\{1,2\}, G$ $\{1,2,3\}$ **in**
  B_1
| |
  B_2
| |
  B_3
endpar

□

12 Unifying the ";" and ">>" operators syntactically

The distinction between the action prefix operator (";") and the enabling operator (">>") is often felt troublesome by novice users and awkward by experienced users. The former is asymmetric, because its left argument cannot be a behaviour expression; the latter is symmetric and associative (unless there are "**accept**" clauses).

Other languages (for instance ESTEREL or Theoretical CSP) have symmetric sequential composition. In his thesis [Bri88], Ed Brinksma addressed this problem and proposed to merge both operators ";" and ">>" from a syntactical point of view. In this section, we present an outline of his proposal.

Basically, the modification consists in replacing the existing syntax of the enabling operator:

$$B_1 \;\texttt{>>}\; [\textbf{accept}\ V_1 : T_1, ..., V_n : T_n\ \textbf{in}]\ B_2$$

with:

$$B_1\ [\textbf{init}\ V_1 : T_1, ..., V_n : T_n]\ \ ;\ \ B_2$$

We make the following comments:

- This syntactic change has several consequences, the ";" operator becomes overloaded; it has two possible profiles, e.g., "$G\ ;\ B_0$" and "$B_1\ ;\ B_2$". This may lead to syntactic ambiguities, which have to be carefully avoided.

  For instance, there is certainly an ambiguity between:

  ```
  <process-identifier> "[" <gate-identifier> "]" ";" <behaviour-expression>
  ```

  and:

  ```
  <gate-identifier> "[" <variable-identifier> "]" ";" <behaviour-expression>
  ```

  A simple solution to this problem would be to prohibit boolean guards in action denotations when no offers are present.

- Also, like the ">>" operator, the ";" operator is not associative, because of value passing and variable scoping. For instance:

  $$B_1 \textbf{ init } V_1 : T_1\ ;\ B_2 \textbf{ init } V_2 : T_2\ ;\ B_3$$

  can be interpreted either as:

  $$B_1 \textbf{ init } V_1 : T_1\ ;\ (B_2 \textbf{ init } V_2 : T_2\ ;\ B_3)$$

  or as:

  $$(B_1 \textbf{ init } V_1 : T_1\ ;\ B_2) \textbf{ init } V_2 : T_2\ ;\ B_3$$

  in which case V_1 is not visible in B_3.

In connection with his proposal for a bracketed syntax (see Section 9 above), Ed Brinksma proposed a syntax to support the unification of the operators ";" and ">>". We give here a synthetic view of this syntax, which intends to solve ambiguities and to make sequential composition be right-associative. We use four different non-terminals *B*, *SeqB*, *NonSeqB*, *Block*:

- The non-terminal *B* denotes the set of all behaviour expressions;
- The non-terminal *SeqB* denotes the set of all behaviour expressions that are sequential composition;
- The non-terminal *NonSeqB* denotes the set of all behaviour expressions that are not sequential composition;
- The non-terminal *Block* denotes the set of "bracketed" behaviour expressions.

These non-terminals are defined using the following grammar (slightly adapted from [Bri88]):

$$\begin{array}{rcl}
B & ::= & \mathit{NonSeqB} \\
 & | & \mathit{SeqB} \\
\mathit{NonSeqB} & ::= & \textbf{stop} \\
 & | & \textbf{exit} \ldots \\
 & | & \textbf{dis } B_1 \texttt{ [> } B_2 \textbf{ enddis}
\end{array}$$

$$\begin{array}{rcl}
 & \mid & \textbf{sel}\ B_1\ \texttt{[]}\ \ldots\ \texttt{[]}\ B_n\ \textbf{endsel} \\
 & \mid & \textbf{par}\ \ldots B_1\ \texttt{||}\ \ldots\ \texttt{||}\ B_n\ \textbf{endpar} \\
 & \mid & \textbf{hide}\ \ldots\ \textbf{in}\ B \\
 & \mid & \textbf{let}\ \ldots\ \textbf{in}\ B \\
 & \mid & P[\ldots](\ldots) \\
\mathit{SeqB} & ::= & G\ldots\ ;\ B \\
 & \mid & \mathit{Block}\ [\textbf{init}\ V_1 : T_n, \ldots, V_n : T_n]\ ;\ B \\
\mathit{Block} & ::= & \mathit{NonSeqB} \\
 & \mid & \textbf{seq}\ \mathit{SeqB}\ \textbf{endseq}
\end{array}$$

This grammar is perhaps simpler to understand if we eliminate the *SeqB* non-terminal:

$$\begin{array}{rcl}
B & ::= & \mathit{NonSeqB} \\
 & \mid & G\ldots\ ;\ B \\
 & \mid & \mathit{Block}\ [\textbf{init}\ V_1 : T_n, \ldots, V_n : T_n]\ ;\ B \\
\mathit{NonSeqB} & ::= & \textit{(* unchanged *)} \\
\mathit{Block} & ::= & \mathit{NonSeqB} \\
 & \mid & \textbf{seq}\ G\ldots\ ;\ B\ \textbf{endseq} \\
 & \mid & \textbf{seq}\ \mathit{Block}\ [\textbf{init}\ V_1 : T_n, \ldots, V_n : T_n]\ ;\ B\ \textbf{endseq}
\end{array}$$

or if we eliminate the *Block* non-terminal:

$$\begin{array}{rcl}
B & ::= & \mathit{NonSeqB} \\
 & \mid & \mathit{SeqB} \\
\mathit{NonSeqB} & ::= & \textit{(* unchanged *)} \\
\mathit{SeqB} & ::= & G\ldots\ ;\ B \\
 & \mid & \mathit{NonSeqB}\ [\textbf{init}\ V_1 : T_n, \ldots, V_n : T_n]\ ;\ B \\
 & \mid & \textbf{seq}\ \mathit{SeqB}\ \textbf{endseq}\ [\textbf{init}\ V_1 : T_n, \ldots, V_n : T_n]\ ;\ B
\end{array}$$

13 Unifying the ";" and ">>" operators semantically

The unification of sequential composition operators proposed by Ed Brinksma (see Section ref 12) is purely syntactic: [Bri88] keeps the existing semantics of the ">>" operator. However, we find a number of problems in the existing semantics.

Even if both sequential composition operators are noted similarly (using ";"), there is still an essential difference between them: the enabling form involves a rendez-vous on the "δ" gate, thus leading to an "**i**" action when the enabling operator is used. On the other hand whereas the action-prefix operator is atomic and does not generate "**i**" actions.

For instance, both expressions "$G\ ;\ B$" and "$(G\ ;\ \textbf{exit})\ ;\ B$" are not strongly equivalent: in the latter, the G action is followed by an "**i**" action.

The generation of "**i**" actions has the unpleasant effect of increasing the size of the Labelled Transition Systems generated from LOTOS programs. This creates. or contributes to, state explosion without any practical benefit from the specifier's point of view. This is a real problem in LOTOS, for which several solutions have been proposed:

- Some tool developers (e.g., Guenther Karjoth and Carl Binding in the LOEWE tool) have chosen to deviate from the LOTOS standard by not generating "**i**" transitions for ">>" operators.
- In other implementations compatible with the LOTOS standard, LOTOS specifiers are often taught not to use the ">>" operator "too much"!

We propose to solve this problem by changing the semantics of the enabling operator, in order to avoid the generation of "**i**" transitions. In this approach, the passing of continuation is atomic in both forms of sequential composition. This proposal goes beyond the syntactic unification described in Section 12 and achieves semantical unification of both forms of sequential composition.

We propose a new dynamic semantics for the enabling operator:

$$\frac{(B_1 \xrightarrow{L} B_1') \ \wedge \ (L \neq \delta)}{(B_1 \ \ ; \ \ B_2) \xrightarrow{L} (B_1' \ \ ; \ \ B_2)}$$

$$\frac{(B_1 \xrightarrow{\delta} \textbf{stop}) \ \wedge \ (B_2 \xrightarrow{L} B_2')}{(B_1 \ \ ; \ \ B_2) \xrightarrow{L} B_2'}$$

or, more generally, if value passing is involved:

$$\frac{(B_1 \xrightarrow{L} B_1') \ \wedge \ (gate(L) \neq \delta)}{(B_1 \ \textbf{init} \ V_1 : T_1, ..., V_n : T_n \ \ ; \ \ B_2) \xrightarrow{L} (B_1' \ \textbf{init} \ V_1 : T_1, ..., V_n : T_n \ \ ; \ \ B_2)}$$

$$\frac{(B_1 \xrightarrow{\delta \ v_1, ..., v_n} \textbf{stop}) \ \wedge \ (\textbf{let} \ V_1 : T_1 = v_1, ..., V_n : T_n = v_n \ \textbf{in} \ B_2 \xrightarrow{L} B_2')}{(B_1 \ \textbf{init} \ V_1 : T_1, ..., V_n : T_n \ \ ; \ \ B_2) \xrightarrow{L} B_2'}$$

Remark
The second rule for sequential composition assumes that B_2 can make a transition labelled with L. What happens if B_2 cannot make such a transition? Using the proposed semantics, we have:

$$\textbf{exit} \ \ ; \ \ \textbf{stop} = \textbf{stop}$$

More generally, for all B, we have:

$$\textbf{exit} \ \ ; \ \ B = B$$

As the following property holds:

$$B \ \ ; \ \ \textbf{exit} = B$$

we infer that "**exit**" is the neutral element of ";". From a mathematical point of view, it is enjoyable that ";" has a neutral element; this is not the case for the ">>" operator of LOTOS. □

This new semantics solves the problems mentioned above and achieves a perfect symmetry between both forms of sequential composition. For instance, the following property holds:

$$(G \ \ ; \ \ \textbf{exit}) \ \ ; \ \ B \ = \ G \ \ ; \ \ B$$

A strong argument for introducing the ";" operator in E-LOTOS relies in the fact that this operator is more primitive (i.e., more general and more expressive) than the existing ">>" operator of LOTOS. Indeed, using the proposed semantics, the existing ">>" operator of LOTOS can be obtained as a shorthand:

$$B_1 \;\texttt{>>}\; [\textbf{accept}\ V_1 : T_1, ..., V_n : T_n\ \textbf{in}]\ B_2 = B_1\ [\textbf{init}\ V_1 : T_1, ..., V_n : T_n]\ \ ;\ \ (\textbf{i}\ \ ;\ \ B_2)$$

Remark
To increase symmetry with the data language, one could replace variable definitions $V_1 : T_1, ..., V_n : T_n$ with a list of patterns $P_1, ..., P_n$. □

14 Introducing exceptions in the behaviour part

There has been some discussion to introduce exceptions in the data part of E-LOTOS. An outline of a SML-based proposal is given in [JGL+95] and a complete proposal is given in [SG95]. We propose here an exception mechanism for the behaviour part, compatible with the one given in [SG95].

In our approach, exceptions are not an entirely new concept by themselves. They are rather an extension of the existing behavioural semantics. Exception identifiers are simply gate identifiers; however, for convenience, we will note them $X, X', X'', X_1, X_2, ...$ instead of $G, G', G'', G_1, G_2, ...$.

We also extend the transition relation defining the behavioural semantics of LOTOS. Besides transitions of the form "$B_1 \xrightarrow{G\ v_1,...,v_n} B_2$", we allow transitions of the form "$B_1 \xrightarrow{X\ v_1,...,v_n\ \diamond} B_2$", where X is an exception identifier and where "$\diamond$" is a special symbol, which is used to distinguish between ordinary rendez-vous and exceptions. Notice that, in our approach, exceptions (as well as gates) can carry typed values.

When we write "$B_1 \xrightarrow{L} B_2$", the label L can be either of the form "$G\ v_1, ..., v_n$", or of the form "$X\ v_1, ..., v_n\ \diamond$". In the latter case, B_2 should always be equal to **stop**.

Semantically, the "*diamond*" sign should be considered as an additional value offer. The existing definition of label equality in LOTOS is kept unchanged: two labels are equal if they have the same gates (or exceptions) and the same offers. This implies that, if two labels are equal, either both of them have the "$\diamond$" symbol, or none of them has this symbol. In particular, this definition is useful for deciding when concurrent processes synchronize.

We extend the definition of behaviour expressions by introducing two new operators:

$$\begin{array}{lcl}
B & ::= & ... \\
 & \mid & \textbf{raise}\ X\ [E_1, ..., E_n] \\
 & \mid & \textbf{trap} \\
 & & \quad B_0 \\
 & & \textbf{handle} \\
 & & \quad X_1\ [V_1^1 : T_1^1, ..., V_{m_1}^1 : T_{m_1}^1] \rightarrow B_1 \\
 & & \quad ... \\
 & & \quad X_n\ [V_1^n : T_1^n, ..., V_{m_n}^n : T_{m_n}^n] \rightarrow B_n \\
 & & \textbf{endtrap}
\end{array}$$

whose semantics is defined by the following rules:

1. raise of an exception:

$$\frac{(eval(E_1) = v_1) \ \wedge \ ... \ \wedge \ (eval(E_n) = v_n)}{\textbf{raise } X \ [E_1, ..., E_n] \xrightarrow{X \ v_1, ..., v_n \diamond} \textbf{stop}}$$

2. normal execution:

$$\frac{B_0 \xrightarrow{G \ v_1, ..., v_n} B_0'}{\left(\begin{array}{l} \textbf{trap} \\ \quad B_0 \\ \textbf{handle} \\ \quad X_1 \ [V_1^1 : T_1^1, ..., V_{m_1}^1 : T_{m_1}^1] \rightarrow B_1 \\ \quad ... \\ \quad X_n \ [V_1^n : T_1^n, ..., V_{m_n}^n : T_{m_n}^n] \rightarrow B_n \\ \textbf{endtrap} \end{array}\right) \xrightarrow{G \ v_1, ..., v_n} \left(\begin{array}{l} \textbf{trap} \\ \quad B_0' \\ \textbf{handle} \\ \quad X_1 \ [V_1^1 : T_1^1, ..., V_{m_1}^1 : T_{m_1}^1] \rightarrow B_1 \\ \quad ... \\ \quad X_n \ [V_1^n : T_1^n, ..., V_{m_n}^n : T_{m_n}^n] \rightarrow B_n \\ \textbf{endtrap} \end{array}\right)}$$

3. catch of an exception:

$$\frac{\begin{array}{c} (B_0 \xrightarrow{X \ v_1, ..., v_p \ \diamond} \textbf{stop}) \ \wedge \\ (\exists i \in \{1, .., n\}) \ (X = X_i) \ \wedge \ (p = m_i) \ \wedge \ (\forall j \in \{1, ..., p\}) \ (type(v_j) = T_j^i) \ \wedge \\ (\textbf{let } V_1^i : T_1^i = v_1, ..., V_p^i : T_p^i = v_p \textbf{ in } B_i \xrightarrow{L} B_i') \end{array}}{\left(\begin{array}{l} \textbf{trap} \\ \quad B_0 \\ \textbf{handle} \\ \quad X_1 \ [V_1^1 : T_1^1, ..., V_{m_1}^1 : T_{m_1}^1] \rightarrow B_1 \\ \quad ... \\ \quad X_n \ [V_1^n : T_1^n, ..., V_{m_n}^n : T_{m_n}^n] \rightarrow B_n \\ \textbf{endtrap} \end{array}\right) \xrightarrow{L} B_i'}$$

4. propagation of an uncaught exception:

$$\frac{\begin{array}{c} (B_0 \xrightarrow{X \ v_1, ..., v_p \ \diamond} \textbf{stop}) \ \wedge \\ (\forall i \in \{1, .., n\}) \ (X \neq X_i) \ \vee \ (p \neq m_i) \ \vee \ (\exists j \in \{1, ..., p\}) \ (type(v_j) \neq T_j^i) \end{array}}{\left(\begin{array}{l} \textbf{trap} \\ \quad B_0 \\ \textbf{handle} \\ \quad X_1 \ [V_1^1 : T_1^1, ..., V_{m_1}^1 : T_{m_1}^1] \rightarrow B_1 \\ \quad ... \\ \quad X_n \ [V_1^n : T_1^n, ..., V_{m_n}^n : T_{m_n}^n] \rightarrow B_n \\ \textbf{endtrap} \end{array}\right) \xrightarrow{X \ v_1, ..., v_p \ \diamond} \textbf{stop}}$$

Remark
We make the following comments on this semantics:

- The "**raise**" operator is the only way to attach a "$\diamond$" symbol to an action. The standard action-prefix operator of LOTOS does not generate "$\diamond$" symbols.

- When an exception is catched, the control transfer is atomic, thus not visible form the outside.
- In non-deterministic choices, the proposed semantics gives no priority to exceptions versus "ordinary" actions. Especially, the following behaviour expression:

  $$G \; ; \; \textbf{stop} \; \texttt{[]} \; \textbf{raise} \; X$$

  cannot be reduced to:

  $$\textbf{raise} \; X$$

- The reason for which gates and exceptions are the same concept relies in the fact that we need sometimes to synchronize several processes on an exception. For instance, one may write:

  $$\textbf{raise} \; X \;\; \texttt{|[}X\texttt{]|} \;\; \textbf{raise} \; X$$

  which is equivalent to:

  $$\textbf{raise} \; X$$

  by applying LOTOS rules for parallel composition. This is especially useful in the case of the "δ" gate/exception (see below), on which all LOTOS parallel composition operators synchronize; the "δ" gate expresses both synchronous termination of concurrent processes and continuation passing.
- The proposed semantics allows both synchronous termination and asynchronous. For instance, the following expression specifies synchronous termination on exception X:

  $$B_1 \;\; \texttt{|[}X\texttt{]|} \;\; B_2$$

  because both behaviours B_1 and B_2 must do "**raise** X" simultaneously to terminate. On the other hand, the following behaviour expression specifies asynchronous termination:

  ```
  trap
     B1 ||| B2
  handle
     X1 → B'1
     X2 → B'2
  endtrap
  ```

  For instance, if one behaviour B_i does "**raise** X_j", then both processes B_1 and B_2 are aborted and the control flow is transferred to B'_j.
- Reciprocally, gates and exceptions cannot be unified completely: it would not be possible to unify "**raise** X" and "X ; **stop** (this is the reason why the "$\diamond$" symbol is necessary to distinguish these two different behaviour expressions). For instance, let's consider the following behaviour expression:

  ```
  trap
     B
  handle
     X' → B'
  endtrap
  ```

Should gates and exceptions be completely unified, if B does a transition labelled with X, then the above behaviour expression would be ambiguous: it would not be possible to decide whether X is a normal action (in which case, the execution of B should continue normally) or whether X is an exception (in which case, the execution of B should be aborted and X propagated to the context outside, since X is not caught by the exception handler).

- The proposed exception mechanism relies on dynamic scoping: if the body of a recursive LOTOS process contains a "**trap**...**endtrap**" operator, the latest exception handler will be used. This is similar to the "**setjmp/longjmp**" mechanism of the C language and different from ADA's statically-scoped exceptions.

□

Semantically speaking, the proposed "**raise**" and "**trap**...**endtrap**" operators are very powerful. They are primitive operators that allow to express several LOTOS operators as derived cases (shorthands):

1. The "**exit**" operator of LOTOS can be defined as:

$$\mathbf{exit}\ [E_1, ..., E_n] =_{def} \mathbf{raise}\ \delta\ [E_1, ..., E_n]$$

2. The ";" operator proposed in Section 13 can be defined as:

$$B_1\ [\mathbf{init}\ V_1 : T_1, ..., V_n : T_n]\ \ ;\ \ B_2 =_{def} \left(\begin{array}{l} \mathbf{trap} \\ \quad B_1 \\ \mathbf{handle} \\ \quad \delta\ [V_1 : T_1, ..., V_n : T_n] \rightarrow B_2 \\ \mathbf{endtrap} \end{array}\right)$$

3. The ">>" operator of LOTOS can be defined as:

$$B_1\ \texttt{>>}\ [\mathbf{accept}\ V_1 : T_1, ..., V_n : T_n\ \mathbf{in}] B_2 =_{def} \left(\begin{array}{l} \mathbf{trap} \\ \quad B_1 \\ \mathbf{handle} \\ \quad \delta\ [V_1 : T_1, ..., V_n : T_n] \rightarrow \mathbf{i}\ \ ;\ \ B_2 \\ \mathbf{endtrap} \end{array}\right)$$

Remark
It is not necessary to hide "δ", because no δ-event can be observed from the outside if the exception is catched. □

4. The "[>" operator of LOTOS can be defined as:

$$B_1\ \texttt{[>}\ B_2 =_{def} \left(\begin{array}{l} \mathbf{trap} \\ \quad B_1\ \texttt{|||}\ (\mathbf{exit}\ \texttt{[]}\ \mathbf{raise}\ \xi) \\ \mathbf{handle} \\ \quad \xi \rightarrow B_2 \\ \mathbf{endtrap} \end{array}\right)$$

where ξ is an exception identifier. This definition deserves a few comments: the exception ξ is not synchronized by the parallel operator and therefore can be spontaneously triggered at any time; if so, the execution of B_1 is aborted and the control flow is transferred to B_2; however, B_1 can also execute normally; if B_1 reaches an "**exit**" statement (also proposed by the right operand of the parallel process), then the δ exception is raised and propagated outside, since it is not caught by the exception handler.

Remark
Omitting the "**exit**" alternative on the right hand-side of the parallel operator would prevent B_1 from terminating successfully, as the "δ" exception is synchronized by parallel composition. □

Remark
It is not necessary to hide "ξ", because no ξ-event can be observed from the outside if the exception is catched. □

Remark
Notice that B_1 could be allowed to raise ξ by itself, thus passing the control to B_2 explicitly. This possibility is also implicitly available in LOTOS if, at some points, the behaviour of B_1 stops (i.e., becomes equivalent to "**stop**"). □

Remark
Of course, the very useful watchdog construct "$(B_1$ `[>` $B_2)$ `>>` B_3" can still be obtained as a particular form of "**trap**". But the "**trap**" operator allows more general forms of watchdogs, in which several events, leading to different behaviours, can be used to escape the watchdog (in LOTOS, only the "δ" event can be used). □

The following remarks are a topic for further work.

Remark
This proposal for introducing exceptions in E-LOTOS should be compared and combined with existing proposals for generalized termination. It should also be compared with [Ber93] who defines the whole semantics of the ESTEREL in terms of preemption (i.e., interrupts). □

Remark
Adding an "**otherwise** $\to B_{n+1}$" clause before the "**endtrap**" keyword could be useful to catch all exceptions but those explicitly listed (i.e., $X_1, ..., X_n$). It would catch any exception, without or with attached values $v_1, ..., v_p$; however, in the latter case, these values could not be referenced in B_{n+1}. □

Remark
The proposed exception mechanism could also be used to catch exceptions generated by value expressions. This would unify exception handling in both the behaviour and data part of E-LOTOS. □

Remark
To increase symmetry with the data language, one could replace variable definitions $V_1^i : T_1, ..., V_{n_i}^i : T_{n_i}$ with a list of patterns $P_1, ..., P_{n_i}$. □

Remark
Constraints could be added to the static semantics of E-LOTOS in order to verify that exceptions are used in a consistent way. For instance:

- Exceptions could be declared together with the types of their parameters. This would be similar to the concept of "gate typing". The case of "overloaded" exceptions (similar to "overloaded" gates) is left for further studies.

- Functionality rules could be extended in order to check statically that exceptions are caught appropriately, with right number and the right types of offers. Currently, functionality rules only deal with the "δ" exception. They should be extended to deal with all exceptions.
- Process declarations could be annotated with a "**raises** $X_1, ..., X_n$" clause indicating which exceptions are raised by a given process. A similar declaration exists in ADA.
- Also, functionality rules could ensure that exceptions are not propagated outside of the scope of their declaration (a problem that arises in SML). This could be done by requiring that exceptions are always caught at the level of their scope unless it is already done in nested scopes.

□

15 Introducing iterators in the behaviour part

Many reactive systems have a cyclical behaviour. In most sequential languages, such behaviours can be described using either iteration or recursion. In LOTOS, however, only recursion is available: all cyclical behaviours have to be described using recursive processes.

We therefore propose to introduce an iterator in E-LOTOS. It is merely a shorthand notation, defined using a recursive process and the exception mechanism defined in Section 14.

We first introduce a new LOTOS operator, whose syntax is:

$$B \quad ::= \quad \textbf{continue}\ [E_1, ..., E_n]$$

and which is equivalent to:

$$\textbf{raise}\ \theta\ [E_1, ..., E_n]$$

where θ is an exception identifier.

We then introduce an iteration operator, whose syntax is:

$$\begin{array}{lll} B & ::= & \textbf{loop}\ [V_1 : T_1\ \texttt{:=}\ E_1, ..., V_n : T_n\ \texttt{:=}\ E_n\ \textbf{in}] \\ & & \quad B_0 \\ & & \textbf{endloop} \end{array}$$

and which is equivalent to:

```
P[...](E_1, ..., E_n)
where
process P[...](V_1 : T_1, ..., V_n : T_n)...
  trap
    B_0
  handle
    θ V'_1 : T_1, ..., V'_n : T_n → P[...](V'_1, ..., V'_n)
  endtrap
endproc
```

In its simplest form, this operator can be used to repeat infinitely a given behaviour. The occurrence of the "**continue**" operator triggers the next iteration. For instance, a simple one-slot buffer accepting two different types of data can be defined as:

```
loop
   INPUT ?V1:DATA_1;
      OUTPUT !V1;
         continue
   []
   INPUT ?V2:DATA_2;
      OUTPUT !V2;
         continue
endloop
```

This operator also allows values to be transmitted from one iteration to the next one. These values are stored in variables $V_1, ..., V_n$ whose initial values are $E_1, ..., E_n$ respectively. For instance, the following cyclical behaviour receives a stream of values X_i on its `INPUT` gate and continuously emits on its `OUTPUT` gate the sum, the minimum, and the maximum of all X_i's received previously:

```
loop SUM:REAL := 0, MIN:REAL := INFINITY, MAX:REAL := MINUS_INFINITY in
   INPUT ?Xi:REAL;
      let NEW_MIN:REAL := if Xi < MIN then Xi else MIN endif in
         let NEW_MAX:REAL := if Xi > MAX then Xi else MAX endif in
            OUTPUT !(SUM + Xi) !NEW_MIN !NEW_MAX;
               continue (SUM + Xi, NEW_MIN, NEW_MAX)
endloop
```

Finally, the "**exit**" operator can be used to go out of the loop. For instance, the following process reads a stream of values on its `INPUT` gate until the sum of these values exceeds 1000 (in which case, it returns the number of values he has read):

```
loop COUNT:NAT := 0, SUM:REAL := 0 in
   INPUT ?Xi:REAL;
      if (SUM + Xi > 1000) then
         exit (COUNT + 1)
      else
         continue (COUNT + 1, SUM + Xi)
endloop
```

Remark
The extended functionality constraints mentioned in Section 14 should ensure that the number and types of values passed to a "**continue**" operator are compatible with the list of variables declared after the "**loop**" keyword of the innermost loop construct. This should be a simple application of general rules for exceptions, rather than rules specifically tailored for loop constructs. □

16 Removing the "where" clause from process definitions

LOTOS processes can contain local definitions of processes and types. These definitions are introduced by the "**where**" keyword. We see a number of drawbacks to this possibility:

- It often obscures LOTOS descriptions, as processes and types can be nested in processes at arbitrary depths.

- It prevents reusability, as types defined in a process are not visible elsewhere and cannot be reused.
- With respect to ActOne, it creates a dissymmetry between the behaviour part and the data type part, as ActOne types cannot contain nested types (nor processes).
- With respect to the data type language proposed for E-Lotos [JGL+95], it creates dissymmetry between the behaviour part and the data type part, as functions cannot contain nested functions (nor processes).

We believe that this possibility should be suppressed and transferred to the module system of E-Lotos. Syntactically, one should replace a process definition having local definitions:

```
process P...
  B
where
  local definitions
endproc
```

with a module having a "hidden" part, introduced by the "**where**" keyword, as in Brinksma's thesis [Bri88] or as in the Lotosphere proposal:

```
module M is
  process P...
    B
  endproc
where
  local definitions
endmod
```

The local process definitions should themselves be "flattened" recursively, in order to eliminate nested processes by putting them altogether at the same level, possibly using appropriate renamings to guarantee unique names.

17 Simplifying process definitions

In order to make a clear symmetry between the behaviour part and the data type language proposed in [SG95], we suggest several changes to the syntax of process definitions, especially with respect to functionality declarations (in standard Lotos, functionality denotes the types of the results returned, using the "**exit**" operator, by a process).

Alan Jeffrey suggested that the declaration of functionality for processes and the declarations of function results should be somehow unified. He proposed to model functionality as an SML tuple type. However, for a number of reasons exposed in [GS95], we believe that using "**out**" parameters for functions, instead of tuples, is highly preferable.

To ensure compatibility with the data type language proposed in [SG95], we suggest to introduce "**out**" parameters in process definitions to replace functionality declarations. We propose to replace process definitions such as:

$$\begin{array}{l}\textbf{process } P[G_1, ..., G_n](V_1 : T_1, ..., V_m : T_m) : \textbf{exit } (T'_0, ..., T'_p) \ := \\ \quad B \\ \textbf{endproc}\end{array}$$

with:

$$\begin{array}{l}\textbf{process } P[G_1, ..., G_n](\textbf{in } V_1 : T_1, ..., \textbf{in } V_m : T_m, \textbf{out } V'_0 : T'_0, ..., \textbf{out } V'_p : T'_p) \textbf{ is} \\ \quad B_0 \\ \textbf{endproc } [P]\end{array}$$

where $V'_0, ..., V'_p$ are new variable names. At this point, three changes should be noticed:

- The "**exit**" clause was replaced with "**in**" and "**out**" attributes. We believe that this new syntax is more compatible with the major languages standardized by ISO/IEC SC22 and also the IDL language of ODP (see [GS95] for a discussion of interoperability).
- The ":=" keyword was replaced with "**is**" according to the syntactic conventions proposed in [SG95].
- The optional facility to recall the name of process P after the "**endproc**" keyword was added. This also exists in ADA and would standardize current practice: many LOTOS specifiers add "*(* P *)*" after "**endproc**" (see for instance the OSI-TP description).

We now consider the cases of process definitions whose functionality is either "**exit**" or "**noexit**". In both cases, we propose to replace such definitions with:

$$\begin{array}{l}\textbf{process } P[G_1, ..., G_n](V_1 : T_1, ..., V_m : T_m) \textbf{ is} \\ \quad B_0 \\ \textbf{endproc } [P]\end{array}$$

We make the following comments:

- All variables V_i could have been declared with the "**in**" attribute, but this is not mandatory since there is no "**out**" attribute.
- The "**noexit**" keyword, which always seems cryptic to new LOTOS users, disappears.
- Above all, we make no distinction between functionalities "**exit**" and "**noexit**". Anyway, the distinction in LOTOS is absolutely meaningless. Functionality rules are designed to protect the specifier against potential mistakes in continuations. However, they address an undecidable problem (the halting problem, precisely) and therefore rely on rough approximations. For instance, the following behaviour expression:

  $$\begin{array}{l}[V] \rightarrow \textbf{stop} \\ [] \\ [\neg V] \rightarrow \textbf{exit}\end{array}$$

  has functionality "**exit**": at 50%, it could have functionality "**noexit**" as well! Similarly, the following behaviour expressions:

  $$[\mathit{false}] \rightarrow \textbf{exit}$$

and:

$$[\mathit{false}] \rightarrow \mathbf{exit}(\mathit{true}, \mathit{true})$$

have functionalities "**exit**" and "**exit** (*bool*, *bool*)" respectively, whereas they are equivalent to "**stop**"!

We propose to keep the functionality rules, but to relax them by getting rid of the subtle distinction between "**exit**" and "**noexit**". By removing this distinction, it will no longer be allowed to make unverifiable statements such as: this behaviour terminates or not. It will also lead to a more user-friendly syntax and a simpler static semantics.

The "**exit**" operator should be slightly modified to take into account the names of the variable declared with the "**out**" attribute. For instance, it should be allowed to write (possibly with a permutation):

$$\mathbf{exit}\ (V'_0\ \texttt{:=}\ E'_0, ..., V'_p\ \texttt{:=}\ E'_p)$$

Note: The treatment of "**exit**" is also closely linked to the problem of gate typing [Gar95], because of the dual nature of "**exit**": it is both a way to return values and also a rendez-vous on the "δ" gate.

18 Abbreviating gate parameters lists

In many Lotos descriptions, process definitions tend to have large lists of gate parameters. This situation has several drawbacks:

- Large lists of gate parameters are tedious to write and difficult to read.
- More often than not, the actual gate parameters of a process instantiation are identical to the formal parameters of the process definition. In such case, actual parameter lists carry no relevant information, but their "syntactic noise" obscures Lotos descriptions.
- Large lists of gate parameters are error-prone. Omitted or extra parameters are detected during static semantics checking. But permuted gate parameters are not, although they introduce subtle semantic errors.
- Finally, adding or deleting a gate parameter from a process P is usually tedious, because it is necessary to modify all instantiations of P, as well as the definitions and instantiations of many processes transitively called by P.

We believe that these problems could often be solved by the adoption of shorthand notations for formal and actual gate parameter lists.

The proposed modifications require the introduction of a new keyword "`...`".

18.1 Abbreviated formal gate parameter lists

Note: this section is technically incompatible with the proposal to remove nested processes made in Section 16 above. It should be ignored if the proposal of Section 16 is accepted.

The definition of non-terminal symbol `<gate-parameter-list>` in the BNF syntax of Lotos should be modified as follows:

```
<gate-parameter-list> ::=
                "[" <gate-identifier-list> "]"
              | "[" ... "]"
              | "[" <gate-identifier-list> "..." "]"
              | "[" "..." <gate-identifier-list> "]"
              | "[" <gate-identifier-list> "..." <gate-identifier-list> "]" ;
```

Remark
This definition is still valid even if there are no formal gate parameters, in which case, according to the syntactic definition of LOTOS, the non-terminal symbol `<gate-parameter-list>` is not used. □

The semantics of an abbreviated formal gate parameter list is simple: if the "`...`" keyword is present in the formal gate parameter list of some process P, this keyword has to be replaced by the list of formal gate parameters of the process containing P (i.e., the smallest process in the definition of which the definition of P is nested). Consequently, this abbreviation is not allowed for the formal gate parameter list of the specification itself.

For instance, the following fragment:

```
process P1 [G1, G2] : noexit :=
   stop
where
   process P2 [G0 ... G3, G4] : noexit :=
      stop
   where
      process P3 [...] : noexit :=
         stop
      endproc
   endproc
endproc
```

is equivalent to:

```
process P1 [G1, G2] : noexit :=
   stop
where
   process P2 [G0, G1, G2, G3, G4] : noexit :=
      stop
   where
      process P3 [G0, G1, G2, G3, G4] : noexit :=
         stop
      endproc
   endproc
endproc
```

18.2 Abbreviated actual gate parameter lists

The definition of non-terminal symbol `<actual-gate-list>` in the BNF syntax of LOTOS should be modified as follows:

```
<actual-gate-list> ::= "[" <gate-identifier-list> "]"
                     | "[" "..." "]"
                     | "[" <gate-substitutions> "]"
```

```
                    |  "[" <gate-substitutions> "..." "]" ;

<gate-substitutions> ::= <gate-substitution>
                     |  <gate-substitution> "," <gate-substitutions> ;

<gate-substitution> ::= <formal-gate> ":=" <actual-gate> ;

<formal-gate> ::= <gate-identifier> ;

<actual-gate> ::= <gate-identifier> ;
```

Remark
This definition is still valid even if there are no actual gate parameters, in which case, according to the syntactic definition of LOTOS, the non-terminal symbol `<actual-gate-list>` is not used. □

The semantics of abbreviated actual gate parameter lists is defined as follows. Let's consider the instantiation of some process P:

1. An `<actual-gate-list>` of the form `"[" <gate-identifier-list> "]"` has the same meaning as in standard LOTOS.

2. An `<actual-gate-list>` of the form `"[" "..." "]"` has to be replaced by the formal gate parameter list of P. For instance, the following fragment:

   ```
   process P1 [G1, G2] : noexit :=
      G1; P1 [...]
      []
      G2; P2 [...]
   endproc
   process P2 [G1, G2] : noexit :=
      G1; G2; P1 [...]
   endproc
   ```

   is equivalent to:

   ```
   process P1 [G1, G2] : noexit :=
      G1; P1 [G1, G2]
      []
      G2; P2 [G1, G2]
   endproc
   process P2 [G1, G2] : noexit :=
      G1; G2; P1 [G1, G2]
   endproc
   ```

3. Let's consider an `<actual-gate-list>` of the form `"[" <gate-substitutions> "]"`. Let $G_0, ..., G_n$ be the formal gate parameter list of P. Then `<gate-substitutions>` must satisfy the following constraint: each G_i must occur once and only once on the left-hand side of a "`:=`" symbol in `<gate-substitutions>`.

   `<actual-gate-list>` has to be replaced by the gate list $G'_0, ..., G'_n$ such that, for each $i \in \{0, ..., n\}$, "G_i `:=` G'_i" belongs to `<gate-substitutions>`.

   Remark
   It is therefore necessary to extend scope rules in order to allow formal gate parameters of LOTOS processes to be visible in process instantiations (on the left-hand side of "`:=`" symbols only). □

Remark
As a consequence of the above replacement rule, all gates occurring on the right-hand side of a "`:=`" symbol in `<gate-substitutions>` must be visible at the point of the Lotos description where P is instantiated. □

Remark
`<gate-substitutions>` determines a total function that maps the formal gate parameters of P onto the actual ones. This function is not necessarily injective: there can exist i_1 and i_2 and a gate G such that `<gate-substitutions>` contains both "G_{i_1} `:=` G" and "G_{i_2} `:=` G". □

For instance, the following fragment:

```
process P1 [G1, G2] : noexit :=
   G1; P1 [G1:=G2, G2:=G1]
   []
   G2; P2 [G1:=G1, G3:=G2]
endproc
process P2 [G1, G3] : noexit :=
   G1; G3; P1 [G1:=G3, G2:=G3]
endproc
```

is equivalent to:

```
process P1 [G1, G2] : noexit :=
   G1; P1 [G2, G1]
   []
   G2; P2 [G1, G2]
endproc
process P2 [G1, G3] : noexit :=
   G1; G3; P1 [G3, G3]
endproc
```

4. Let's consider an `<actual-gate-list>` of the form `"[" <gate-substitutions> "..." "]"`. Let $G_0, ..., G_n$ be the formal gate parameter list of P. Then `<gate-substitutions>` must satisfy the following constraint: each G_i may occur at most once on the left-hand side of a "`:=`" symbol in `<gate-substitutions>`.

   `<actual-gate-list>` has to be replaced by the gate list $G'_0, ..., G'_n$ such that, for each $i \in \{0, ..., n\}$, either "G_i `:=` G'_i" belongs to `<gate-substitutions>`, or[4] "$G'_i = G_i$".

   Remark
   `<gate-substitutions>` determines a partial function that maps the formal gate parameters of P onto the actual ones (*explicit parameters*). All formal gates not mentioned in `<gate-substitutions>` are kept unchanged (*implicit parameters*). This function is not necessarily injective. □

   Remark
   The "`...`" symbol is allowed even if `<gate-substitutions>` contains as many substitutions as the number of formal gate parameters of P, i.e. even if all actual parameters are explicit parameters. □

   For instance, the following fragment:

[4]this is an exclusive "or"

```
    process P1 [G1, G2] : noexit :=
       G1; P1 [G1:=G2 ...]
       []
       G2; P2 [G3:=G2 ...]
    endproc
    process P2 [G1, G3] : noexit :=
       G1; G3; P1 [G2:=G3 ...]
    endproc
```

is equivalent to:

```
    process P1 [G1, G2] : noexit :=
       G1; P1 [G2, G2]
       []
       G2; P2 [G1, G2]
    endproc
    process P2 [G1, G2] : noexit :=
       G1; G3; P1 [G1, G3]
    endproc
```

The proposed modification is upward compatible, except for those existing LOTOS descriptions that contain operation identifiers with the same spelling as the new reserved keyword "...". For those descriptions, renaming the conflicting identifiers would be needed.

Reciprocally, any LOTOS description with abbreviated gate parameter lists can be translated into standard LOTOS by expanding the "..." symbols.

Remark
The proposed modification fits well with another proposal for the introduction of typed gates in LOTOS [Gar94a]. The syntactic notations and underlying semantics are similar in both proposals. □

Remark
An alternative approach for abbreviating gate parameter lists would be the possibility to define identifiers for (formal and actual) gate parameter lists. These identifiers could be used in place of the "..." notation. It is not clear, however, if this alternative approach is worth its complexity and if it can be extended to value parameter lists (see next section) and incomplete action denotations [Gar94a]. □

19 Abbreviating value parameters lists

Similarly, it is desirable to shorten the large list of value parameters. This can be achieved with the same mechanism as the one proposed for gate parameters. The only difference comes from the fact that formal parameters are value identifiers whereas actual parameters are value expressions.

Therefore, only the proposed new syntax is given, together with examples illustrating the use of the abbreviated constructions.

19.1 Abbreviated formal value parameter lists

Note: this section is technically incompatible with the proposal to remove nested processes made in Section 16 above. It should be ignored if the proposal of Section 16 is accepted.

The proposed modified syntax is the following:

```
<value-parameter-list> ::=
       "(" <identifier-declarations> ")"
    |  "(" "..." ")"
    |  "(" <identifier-declarations> "..." ")"
    |  "(" "..." <identifier-declarations> ")"
    |  "(" <identifier-declarations> "..." <identifier-declarations> ")" ;
```

For instance, the following fragment:

```
process P1 [G] (X1 : BOOL, X2 : NAT) : noexit :=
   stop
where
   process P2 [G] (X0 : NAT ... X3 : BOOL) : noexit :=
      stop
   where
      process P3 [G] (...) : noexit :=
         stop
      endproc
   endproc
endproc
```

is equivalent to:

```
process P1 [G] (X1 : BOOL, X2 : NAT) : noexit :=
   stop
where
   process P2 [G] (X0 : NAT, X1 : BOOL, X2 : NAT, X3 : BOOL) : noexit :=
      stop
   where
      process P3 [G] (X0 : NAT, X1 : BOOL, X2 : NAT, X3 : BOOL) : noexit :=
         stop
      endproc
   endproc
endproc
```

19.2 Abbreviated actual value parameter lists

The proposed modified syntax is the following:

```
<actual-parameter-list> ::= "(" <value-expression-list> ")"
                         |  "(" "..." ")"
                         |  "(" <value-substitutions> ")"
                         |  "(" <value-substitutions> "..." ")" ;

<value-substitutions> ::= <value-substitution>
                       |  <value-substitution> "," <value-substitutions> ;

<value-substitution> ::= <formal-value> ":=" <actual-value> ;

<formal-value> ::= <value-identifier> ;
```

```
   <actual-value> ::= <value-expression> ;
```

For instance, the following fragment:

```
   process P [G] (X, Y : NAT) :=
      [X < 10] ->
         P [G] (X := X + 1 ...)
      []
      [(X >= 10) and (Y < 10)] ->
         P [G] (Y := Y + 1 ...)
      []
      [(X >= 10) and (Y >= 10)] ->
         P [G] (X := 0, Y := 0)
   endproc
```

is equivalent to:

```
   process P [G] (X, Y : NAT) :=
      [X < 10] ->
         P [G] (X + 1, Y)
      []
      [(X >= 10) and (Y < 10)] ->
         P [G] (X, Y + 1)
      []
      [(X >= 10) and (Y >= 10)] ->
         P [G] (0, 0)
   endproc
```

Remark
The proposed abbreviated notation introduces an assignment notation (using the "`:=`" symbol) that carries, more or less, the usual meaning of assignment. This proves to be useful when translating into LOTOS some descriptions written in languages with explicit assignments (e.g., SDL [CCI88] or ESTELLE [ISO88a]).

It is to be mentioned that the assignment notation is merely a syntactic facility and does not subvert the semantics of LOTOS as a functional language. □

Remark
Compared to the existing process instantiation in standard LOTOS, the proposed abbreviation has one major advantage: it lays the emphasis on "what is changing" and indicates clearly which variables are modified. □

The benefits of the two improvements proposed in Sections 18 and 19 are demonstrated in Annexes B and C.

Conclusion

Nineteen changes have been proposed to improve the behaviour part of LOTOS.

Some of these changes only concern syntactic and static semantic aspects: most of them are easy to implement and fully upward compatible; the others are upward compatible if identifiers in conflict with new keywords are renamed.

Some other changes also affect the existing dynamic semantics, especially when new operators are introduced. We believe that the advantages of the proposed improvements suffice to justify the change from existing LOTOS.

All these proposals have been presented more or less independently. Further work is needed to integrate them together.

Acknowledgements

Acknowledgements are due to Arnaud Février, Alain Kerbrat, Laurent Mounier, Elie Najm and Jacques Sincennes for their useful comments on version 1.0 of this document.

Acknowledgements are due to Radu Mateescu for his useful comments on version 2.0 of this document.

References

[Ber93] Gérard Berry. Preemption and Concurrency. In *Proceedings of FSTTCS 93*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93, Berlin, 1993. Springer Verlag.

[Bri88] Ed Brinksma. *On the Design of Extended LOTOS, a Specification Language for Open Distributed Systems.* PhD thesis, University of Twente, November 1988.

[CCI88] CCITT. Specification and Description Language. Recommendation Z.100, International Consultative Committee for Telephony and Telegraphy, Genève, March 1988.

[Gar94a] Hubert Garavel. On the Introduction of Gate Typing in E-LOTOS. Rapport SPECTRE 94-3, VERIMAG, Grenoble, February 1994. Annex D of ISO/IEC JTC1/SC21/WG1 N1314 Revised Draft on Enhancements to LOTOS and Annex C of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[Gar94b] Hubert Garavel. Six improvements to the process part of LOTOS. Rapport SPECTRE 94-7, VERIMAG, Grenoble, June 1994. Annex K of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[Gar95] Hubert Garavel. On the Introduction of Gate Typing in E-LOTOS. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP International Workshop on Protocol Specification, Testing and Verification (Warsaw, Poland)*, London, June 1995. IFIP, Chapman & Hall.

[GS95] Hubert Garavel and Mihaela Sighireanu. French-Romanian Comments regarding some Proposed Features for E-LOTOS Data Types. Rapport SPECTRE 95-19, VERIMAG, Grenoble, December 1995. Input document [xxx] of the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

[ISO88a] ISO/IEC. ESTELLE — A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[ISO88b] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for

Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[JGL$^+$95] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21/WG7 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[SG95] Mihaela Sighireanu and Hubert Garavel. A Proposal for the Data Type Part of E-LOTOS Applicable to the Formal Description of OSI and ODP Standards. Rapport SPECTRE 95-20, VERIMAG, Grenoble, December 1995. Input document [xxx] of the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

A Expressing the "if" operator in standard LOTOS

We give here a concrete syntax for the "**if**" operator defined in Section 5:

```
<behaviour-expression> ::=
  "if" <value-expression> "then" <behaviour-expression>
  [ "elsif" <value-expression> "then" <behaviour-expression> ]*
  [ "else" <behaviour-expression> ]
  "endif"
```

where "[...]∗" denotes a repeated occurrence zero or more times (meaning that there can be zero or more "**elsif**" clauses) and where "[...]" denotes an optional occurrence (meaning that the "**else**" clause is optional). An equivalent syntactic definition is the following:

```
<behaviour-expression> ::=
      "if" <value-expression> "then" <behaviour-expression> <elsif-part-list> ;

<elsif-part-list> ::= <elsif-part> <elsif-part-list>
                   |  <else-part> ;

<elsif-part> ::= "elsif" <value-expression> "then" <behaviour-expression> ;

<else-part> ::= "else" <behaviour-expression> <endif-part>
             |  <endif-part> ;

<endif-part> ::= "endif" ;
```

The semantics of the "**if**" operator is expressed using a transformation function $[\![.]\!]$ that expands "**if**" operators into combinations of guards and non-deterministic choices. Therefore, any description with "**if**" constructs can be translated into standard LOTOS using a macro-processor that implements the expansion function $[\![.]\!]$. This function is defined as follows. If B is a behaviour expression of the form:

```
if $E_0$ then $B_0$
elsif $E_1$ then $B_1$
elsif $E_2$ then $B_2$
...
elsif $E_n$ then $B_n$
else $B_{n+1}$
endif
```

then $[\![B]\!]$ is equal to:

```
(
[$E_0$] -> ($[\![B_0]\!]$)
[]
[not ($E_0$) and ($E_1$)] -> ($[\![B_1]\!]$)
[]
[not ($E_0$) and not ($E_1$) and ($E_2$)] -> ($[\![B_2]\!]$)
[]
...
[]
[not ($E_0$) and not ($E_1$) and not ($E_2$) ... and not ($E_{n-1}$) and ($E_n$)] -> ($[\![B_n]\!]$)
[]
[not ($E_0$) and not ($E_1$) and not ($E_2$) ... and not ($E_{n-1}$) and not ($E_n$)] -> ($[\![B_{n+1}]\!]$)
)
```

Remark
If the "**elsif**" and/or "**else**" parts are missing in B, the corresponding expanded parts have to be removed from $[\![B]\!]$. □

This expansion scheme is not optimal because it generates multiple occurrences of expressions $E_0, ..., E_n$, therefore leading to multiple evaluations of the same expressions[5].

A better expansion scheme is shown below. It stores the results of guard evaluation into $(n + 1)$ boolean variables $X_0, ..., X_n$ (the names of which being different from the names of all free variables contained in $E_0, ..., E_n$ and $B_0, ..., B_{n+1}$). In this improved scheme, $[\![B]\!]$ is equal to:

[5]Unless LOTOS tools are smart enough to optimize those situations, which does not seem to be the case now...

```
(
  let $X_0$:bool = $E_0$ in
    (
    [$X_0$] -> ($[\![B_0]\!]$)
    []
    [not ($X_0$)] ->
      let $X_1$:bool = $E_1$ in
        (
        [$X_1$] -> ($[\![B_1]\!]$)
        []
        [not ($X_1$)] ->
          let $X_2$:bool = $E_2$ in
            (
            [$X_2$] -> ($[\![B_2]\!]$)
            []
            [not ($X_2$)] ->
              ...
                      let $X_n$:bool = $E_n$ in
                        (
                        [$X_n$] -> ($[\![B_n]\!]$)
                        []
                        [not ($X_n$)] -> ($[\![B_{n+1}]\!]$)
                        )
              ...
            )
        )
    )
)
```

Remark
There is another, equivalent way to define the improved expansion function $[\![.]\!]$. The expansion can be performed in two successive steps:

- **Step 1:** "**if**" operators with "**elsif**" parts are expanded into nested "**if**" operators without "**elsif**" part, i.e. the following behaviour expression:

```
if $E_0$ then $B_0$
elsif $E_1$ then $B_1$
elsif $E_2$ then $B_2$
...
elsif $E_n$ then $B_n$
else $B_{n+1}$
endif
```

  is expanded into:

```
if $E_0$ then $B_0$
else
   if $E_1$ then $B_1$
   else
      if $E_2$ then $B_2$
      else
         ...
            if $E_n$ then $B_n$
            else $B_{n+1}$
            endif
         ...
      endif
   endif
endif
```

- **Step 2:** "**if**" operators without "**elsif**" part are translated into guarded commands, i.e. the following behaviour expression:

  ```
  if $E$ then $B_1$
  else $B_2$
  endif
  ```

  is expanded into:

  ```
  (
  let $X$:bool = $E$ in
     (
     [$X$] -> ([[$B_1$]])
     []
     [not ($X$)] -> ([[$B_2$]])
     )
  )
  ```

□

B A simplified transport service

The example below is a highly simplified description of a transport service written in Basic LOTOS. The original description is given first, followed by a much more concise description making use of abbreviated gate parameter lists (see Section 18).

```
specification TRANSPORT_SERVICE [A_CONREQ, A_CONIND, A_CONRESP, A_CONCONF, A_DISREQ, A_DISIND,
    B_CONREQ, B_CONIND, B_CONRESP, B_CONCONF, B_DISREQ, B_DISIND] : noexit
behaviour

hide CR, CI, DR, DI in
   (
   TRANSPORT_ENTITY [A_CONREQ, A_CONIND, A_CONRESP, A_CONCONF, A_DISREQ, A_DISIND, CR, CI, DR, DI]
   |[CR, CI, DR, DI]|
   TRANSPORT_ENTITY [B_CONREQ, B_CONIND, B_CONRESP, B_CONCONF, B_DISREQ, B_DISIND, CI, CR, DI, DR]
   )
where

   process TRANSPORT_ENTITY [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI] : noexit :=
   IDLE [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
   where

      process IDLE [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI] : noexit :=
      CONREQ;
         CR;
            (
            WAIT [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
            []
            CI;
               CONCONF;
                  OPEN [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
            )
      []
      CI;
         CONIND;
            (
            WAIT [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
            []
            CONRESP;
               CR;
                  OPEN [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
            )
      endproc

      process WAIT [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI] : noexit :=
      DISREQ;
         DR;
            FROZEN [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
      []
      DI;
         DISIND;
            DR;
               IDLE [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
      endproc

      process OPEN [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI] : noexit :=
      WAIT [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
      endproc

      process FROZEN [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI] : noexit :=
      CI;
         FROZEN [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
      []
      DI;
         IDLE [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI]
      endproc
   endproc
endspec
```

```
specification TRANSPORT_SERVICE [A_CONREQ, A_CONIND, A_CONRESP, A_CONCONF, A_DISREQ, A_DISIND,
    B_CONREQ, B_CONIND, B_CONRESP, B_CONCONF, B_DISREQ, B_DISIND] : noexit
behaviour

hide CR, CI, DR, DI in
   (
   TRANSPORT_ENTITY [A_CONREQ, A_CONIND, A_CONRESP, A_CONCONF, A_DISREQ, A_DISIND, CR, CI, DR, DI]
   |[CR, CI, DR, DI]|
   TRANSPORT_ENTITY [B_CONREQ, B_CONIND, B_CONRESP, B_CONCONF, B_DISREQ, B_DISIND, CI, CR, DI, DR]
   )
where

   process TRANSPORT_ENTITY [CONREQ, CONIND, CONRESP, CONCONF, DISREQ, DISIND, CR, CI, DR, DI] : noexit :=
   IDLE [...]
   where

      process IDLE [...] : noexit :=
      CONREQ;
         CR;
            (
            WAIT [...]
            []
            CI;
               CONCONF;
                  OPEN [...]
            )
      []
      CI;
         CONIND;
            (
            WAIT [...]
            []
            CONRESP;
               CR;
                  OPEN [...]
            )
      endproc

      process WAIT [...] : noexit :=
      DISREQ;
         DR;
            FROZEN [...]
      []
      DI;
         DISIND;
            DR;
               IDLE [...]
      endproc

      process OPEN [...] : noexit :=
      WAIT [...]
      endproc

      process FROZEN [...] : noexit :=
      CI;
         FROZEN [...]
      []
      DI;
         IDLE [...]
      endproc
   endproc
endspec
```

C A simplified sliding window protocol

The example below is a simplified sliding window protocol. For conciseness purpose, the abstract data type definitions are omitted. The original description is given first, followed by a shorter description making use of "**if**" constructs, abbreviated gate parameter lists and abbreviated value parameter lists (see Sections 5, 18, and 19).

```
specification SLIDING_WINDOW_PROTOCOL [PUT, GET] : noexit
behaviour
   hide SDT, RDT, RACK, SACK in
      (
         (
         TRANSMITTER [PUT, SDT, SACK] (ZERO)
         |||
         RECEIVER [GET, RDT, RACK] (ZERO)
         )
      |[SDT, RDT, RACK, SACK]|
         (
         LINE [SDT, RDT] (EMPTY)
         |||
         LINE [RACK, SACK] (EMPTY)
         )
      )
where

process LINE [INPUT, OUTPUT] (R:REG) : noexit :=
   INPUT ?N:NUM;
      (
      LINE [INPUT, OUTPUT] (INSERT (R, N))
      []
      LINE [INPUT, OUTPUT] (SHIFT (R))
      )
   []
      (
      choice E:ELM []
         (
         let N:XNUM = VALUE (R, E) in
            [not (VOID (N))] ->
               OUTPUT !(CONV (N));
                  (
                  LINE [INPUT, OUTPUT] (DELETE (R, E))
                  []
                  LINE [INPUT, OUTPUT] (R)
                  )
         )
      )
endproc

process TRANSMITTER [PUT, SDT, SACK] (BASE:NUM) : noexit :=
   TRANSMIT [PUT, SDT, SACK] (BASE, 0)
where

   process TRANSMIT [PUT, SDT, SACK] (BASE:NUM, SIZE:NAT) : noexit :=
      [SIZE < TWS] ->
         PUT !(BASE + SIZE);
            SDT !(BASE + SIZE);
               TRANSMIT [PUT, SDT, SACK] (BASE, SIZE + 1)
      []
      SACK ?N:NUM;
         (
         let OK:BOOL = WINDOW (N, BASE, SIZE) in
```

```
            (
            [OK] ->
               TRANSMIT [PUT, SDT, SACK] (N + 1, SIZE - ((N + 1) - BASE))
            []
            [not (OK)] ->
               TRANSMIT [PUT, SDT, SACK] (BASE, SIZE)
            )
         )
      []
         (
         choice N:NUM []
            [WINDOW (N, BASE, SIZE)] ->
               i;
                  RETRANSMIT [PUT, SDT, SACK] (N, SIZE - (N - BASE))
         )
   endproc

   process RETRANSMIT [PUT, SDT, SACK] (BASE:NUM, SIZE:NAT) : noexit :=
      [SIZE > 0] ->
         SDT !BASE;
            RETRANSMIT [PUT, SDT, SACK] (BASE + 1, SIZE - 1)
      []
      [SIZE == 0] ->
         TRANSMIT [PUT, SDT, SACK] (BASE, SIZE)
   endproc
endproc

process RECEIVER [GET, RDT, RACK] (BASE:NUM) : noexit :=
   RECEIVE [GET, RDT, RACK] (BASE, RESET)
where

   process RECEIVE [GET, RDT, RACK] (BASE:NUM, RECEIVED:TAB) : noexit :=
      RDT ?N:NUM;
         (
         let OK:BOOL = not (TEST (RECEIVED, N)) and WINDOW (N, BASE, RWS) in
            (
            [OK] ->
               DELIVER [GET, RDT, RACK] (BASE, SET (RECEIVED, N))
            []
            [not (OK)] ->
               RACK !(ZERO + (BASE - ONE));
                  RECEIVE [GET, RDT, RACK] (BASE, RECEIVED)
         )
   endproc

   process DELIVER [GET, RDT, RACK] (BASE:NUM, RECEIVED:TAB) : noexit :=
      let OK:BOOL = TEST (RECEIVED, BASE) in
         (
         [OK] ->
            GET !BASE;
               DELIVER [GET, RDT, RACK] (BASE + 1, UNSET (RECEIVED, BASE))
         []
         [not (OK)] ->
            RACK !(ZERO + (BASE - ONE));
               RECEIVE [GET, RDT, RACK] (BASE, RECEIVED)
         )
   endproc
endproc

endspec
```

```
specification SLIDING_WINDOW_PROTOCOL [PUT, GET] : noexit
behaviour
   hide SDT, RDT, RACK, SACK in
      (
         (
         TRANSMITTER [PUT, SDT, SACK] (ZERO)
         |||
         RECEIVER [GET, RDT, RACK] (ZERO)
         )
      |[SDT, RDT, RACK, SACK]|
         (
         LINE [SDT, RDT] (EMPTY)
         |||
         LINE [RACK, SACK] (EMPTY)
         )
      )
where

process LINE [INPUT, OUTPUT] (R:REG) : noexit :=
   INPUT ?N:NUM;
      (
      LINE [...] (R := INSERT (R, N))
      []
      LINE [...] (R := SHIFT (R))
      )
   []
      (
      choice E:ELM []
         (
         let N:XNUM = VALUE (R, E) in
            [not (VOID (N))] ->
               OUTPUT !(CONV (N));
                  (
                  LINE [...] (R := DELETE (R, E))
                  []
                  LINE [...] (...)
                  )
         )
      )
endproc

process TRANSMITTER [PUT, SDT, SACK] (BASE:NUM) : noexit :=
   TRANSMIT [...] (BASE, 0)
where

   process TRANSMIT [...] (... SIZE:NAT) : noexit :=
      [SIZE < TWS] ->
         PUT !(BASE + SIZE);
            SDT !(BASE + SIZE);
               TRANSMIT [...] (SIZE := SIZE + 1 ...)
      []
      SACK ?N:NUM;
         if WINDOW (N, BASE, SIZE) then
            TRANSMIT [...] (N := N + 1, SIZE := SIZE - ((N + 1) - BASE))
         else
            TRANSMIT [...] (...)
         endif
      []
         (
         choice N:NUM []
            [WINDOW (N, BASE, SIZE)] ->
               i;
```

```
                  RETRANSMIT [...] (BASE := N, SIZE := SIZE - (N - BASE))
         )
   endproc

   process RETRANSMIT [...] (... SIZE:NAT) : noexit :=
      [SIZE > 0] ->
         SDT !BASE;
            RETRANSMIT [...] (BASE := BASE + 1, SIZE := SIZE - 1)
      []
      [SIZE == 0] ->
         TRANSMIT [...] (...)
   endproc
endproc

process RECEIVER [GET, RDT, RACK] (BASE:NUM) : noexit :=
   RECEIVE [...] (BASE, RESET)
where

   process RECEIVE [...] (... RECEIVED:TAB) : noexit :=
      RDT ?N:NUM;
         if not (TEST (RECEIVED, N)) and WINDOW (N, BASE, RWS) then
            DELIVER [...] (BASE, SET (RECEIVED, N))
         else
            RACK !(ZERO + (BASE - ONE));
               RECEIVE [...] (...)
         endif
   endproc

   process DELIVER [...] (... RECEIVED:TAB) : noexit :=
      if TEST (RECEIVED, BASE) then
         GET !BASE;
            DELIVER [...]  (BASE := BASE + 1, RECEIVED := UNSET (RECEIVED, BASE))
      else
          RACK !(ZERO + (BASE - ONE));
             RECEIVE [...] (...)
      endif
   endproc
endproc

endspec
```