Six improvements to the process part of LOTOS

Version 1.0

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`

June, 30, 1994

Abstract

This paper proposes six changes to the syntax and static semantics of the process part of LOTOS. These changes are intended to make LOTOS a more comfortable language; they are almost upward compatible, easy to implement, and do not require in-depth modifications of existing LOTOS tools.

Introduction

This paper proposes six modifications of the Formal Description Technique LOTOS (ISO standard 8807 [ISO88b]). These modifications should simplify the tasks of writing and reading LOTOS descriptions. The proposed modifications only affect the syntax and static semantics of the process part of LOTOS. Dynamic semantics and abstract data types are not addressed here. The proposed changes are listed below by increasing complexity and will be presented in the following sections:

- (C1) Giving a printable name to the "δ" gate
- (C2) Turning the specification identifier into an ordinary process identifier
- (C3) Turning the reserved keyword "**i**" into a predefined gate identifier
- (C4) Introducing an "**if**" operator
- (C5) Abbreviating gate parameters lists
- (C6) Abbreviating value parameters lists

[*]This work has been supported in part by the Commission of the European Communities, under ESPRIT EC-Canada Exploratory Collaborative Activity EC-CA 001:76099 "EUCALYPTUS: A European/Canadian LOTOS Protocol Tool Set".

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 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 "**behaviour**") 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 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 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.

[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

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:

```
<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. This function is defined as follows. If B is a behaviour expression of the form:

$$\begin{array}{l}\textbf{if } V_0 \textbf{ then } B_0\\ \textbf{elsif } V_1 \textbf{ then } B_1\\ \textbf{elsif } V_2 \textbf{ then } B_2\\ \ldots\\ \textbf{elsif } V_n \textbf{ then } B_n\\ \textbf{else } B_{n+1}\\ \textbf{endif}\end{array}$$

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

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

```
(
[$V_0$] -> ($[\![B_0]\!]$)
[]
[not ($V_0$) and ($V_1$)] -> ($[\![B_1]\!]$)
[]
[not ($V_0$) and not ($V_1$) and ($V_2$)] -> ($[\![B_2]\!]$)
[]
...
[]
[not ($V_0$) and not ($V_1$) and not ($V_2$) ... and not ($V_{n-1}$) and ($V_n$)] -> ($[\![B_n]\!]$)
[]
[not ($V_0$) and not ($V_1$) and not ($V_2$) ... and not ($V_{n-1}$) and not ($V_n$)] -> ($[\![B_{n+1}]\!]$)
)
```

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 $V_0, ..., V_n$, therefore leading to multiple evaluations of the same expressions[3].

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 $V_0, ..., V_n$ and $B_0, ..., B_{n+1}$). In this improved scheme, $[\![B]\!]$ is equal to:

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

```
(
  let $X_0$:bool = $V_0$ in
    (
    [$X_0$] -> ($[\![B_0]\!]$)
    []
    [not ($X_0$)] ->
      let $X_1$:bool = $V_1$ in
        (
        [$X_1$] -> ($[\![B_1]\!]$)
        []
        [not ($X_1$)] ->
          let $X_2$:bool = $V_2$ in
            (
            [$X_2$] -> ($[\![B_2]\!]$)
            []
            [not ($X_2$)] ->
              ...
                    let $X_n$:bool = $V_n$ 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** $V_0$ **then** $B_0$
**elsif** $V_1$ **then** $B_1$
**elsif** $V_2$ **then** $B_2$
...
**elsif** $V_n$ **then** $B_n$
**else** $B_{n+1}$
**endif**
```

is expanded into:

```
if $V_0$ then $B_0$
else
  if $V_1$ then $B_1$
  else
    if $V_2$ then $B_2$
    else
      ...
          if $V_n$ 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 $V$ then $B_1$
else $B_2$
endif
```

is expanded into:

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

□

This extension is upward compatible, except for those existing Lotos descriptions that contain identifiers with the same spelling as the new reserved keywords: "**if**", "**then**", "**else**", "**elsif**", and "**endif**". For those descriptions, renaming the conflicting identifiers would be needed.

Reciprocally, any Lotos description with "**if**" constructs can be translated into standard Lotos using a macro-processor that implements the expansion function $[\![.]\!]$.

5 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 "`...`".

5.1 Abbreviated formal gate parameter lists

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

is equivalent to:

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

5.2 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-substitutions> ::= <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 size 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:

```
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 [Gar94]. 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 [Gar94]. □

[4]this is an exclusive "or"

6 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.

6.1 Abbreviated formal value parameter lists

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

is equivalent to:

```
process P1 [G] (X1 : BOOL, X2 : NAT) : nexit :=
   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
```

6.2 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-substitutions> ::= <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. □

Conclusion

Six changes (C1)–(C6) have been proposed to improve the process part of LOTOS. These changes only concern syntactic and static semantic aspects: the existing dynamic semantics is preserved. These changes are easy to implement. Most of them are totally upward compatible; the others are upward compatible if identifiers in conflict with new keywords are renamed.

The benefits of improvements (C4)–(C6) are demonstrated in Annexes A and B.

Acknowledgements

Acknowledgements are due to Arnaud Février, Alain Kerbrat, Laurent Mounier, Elie Najm and Jacques Sincennes for their useful comments.

References

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

[Gar94] 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.

[ISO88a] ISO. 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. 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.

7 Annex A: 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 (improvement (C4)).

```
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
```

Annex B: 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 (improvements (C4), (C5), and (C6)).

```
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
```