On the Introduction of Gate Typing in E-LOTOS

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

Abstract

The definition of the Formal Description Technique LOTOS (ISO standard 8807) is currently under revision. This paper proposes a gate typing extension to LOTOS in order to improve the current situation where gates are completely typeless. This extension is simple and fully upward compatible. It is shown to increase both the reliability and modularity of formal descriptions. Moreover, gate type-checking can be performed statically and does not require any change in the dynamic semantics of LOTOS.

Introduction

The ISO specification language LOTOS [ISO88] [BB88] has been standardized as a mean to describe protocols and services formally. LOTOS combines sound semantic concepts (borrowed the theories of algebraic abstract data types and process algebras) with powerful language features intended for the design of complex systems. It provides a rich set of specification styles and robust tools are now available, which support design, verification and code generation.

LOTOS has been widely used to describing formally data communication systems. However, feedback from the users has indicated that the usefulness of the language could be improved by improving both its theoretical expressiveness as practical user-friendliness.

For these reasons, LOTOS is currently under revision in ISO ; this revision process should produce a enhanced standard language named E-LOTOS (for Extended LOTOS). Several working documents have been successively produced [Jua93, Jua94a, Jua94b], in which two types of enhancements are considered:

- Some enhancements propose to increase the expressiveness of LOTOS by introducing new features which are currently missing in the language. For instance, it is agreed to introduce a notion of quantitative time in LOTOS in order to allow the description of new classes of protocols and services, e.g., high-speed protocols for multimedia applications.

- Some other enhancements attempt to correct various shortcomings of LOTOS. For instance, it

*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”.

is suggested to allow the description of data structures using a functional language instead of algebraic abstract data types. It is also proposed to introduce features, such as modules, which have proved to be useful for software engineering purpose.

The enhancement described in this paper belongs to this second category. It addresses a recognized drawback of existing LOTOS, in which the interaction points (*gates*) are completely untyped. A mechanism is presented, which allows LOTOS gates to be typed and the corresponding type-checking algorithm is formally defined.

This paper is organized as follows. Section 1 recalls some basic definitions related to LOTOS. Section 2 exposes the main motivations and design choices for the introduction of gate typing in LOTOS. Section 3 presents the proposed enhancement at the lexical, syntactic and static semantics level; the rules defining gate type equivalence and well-typed action denotations are given. Section 4 demonstrates how the proposed gate typing mechanism can significantly improve the modularity of process definitions, especially in the case of constraint-oriented descriptions. Finally, the current status of this proposal in the E-LOTOS standardization process is discussed, and possible extensions are mentioned.

1 Basic definitions

A basic knowledge of LOTOS is assumed. The following terms will be used in the paper:

- An "experiment offer" (or "offer") denotes either a value emission "!V" or a receipt "?$X : S$". It corresponds to the non-terminal symbol `<experiment-offer>` in the syntax of LOTOS.
- An "experiment offer list" is a (possibly empty) list of experiment offers proposed for a rendez-vous. It corresponds to the non-terminal symbol `<experiment-offer-list>`.
- An "action denotation" denotes a rendez-vous proposal. It consists of a gate followed by an experiment offer list, and possibly a boolean guard. (called the "selection predicate"). It corresponds to the non-terminal symbol `<action-denotation>`.
- A "profile" $\langle S_1, ..., S_n \rangle$ is an n-tuple ($n \geq 0$) of sorts, corresponding to the sorts of the experiment offers in an experiment offer list. For instance, the profile of "`G !true !0 ?X,Y:S`" is likely to be "$\langle$`bool`, `nat`, `S`, `S`$\rangle$". This definition of profile will be extended in section 3.

2 Rationale

In LOTOS, gates are untyped; they can accept any profile, i.e., any number of experiment offers, and of any sort.

In its most general definition, a "gate typing" mechanism would consist in associating to each gate declaration a set of constraints that restrict the profiles accepted by the gate.

Gate typing exists in other Formal Description Techniques such as ESTELLE ("channels") and SDL ("signals").

This proposal for introducing gate typing in E-LOTOS is motivated by the following considerations:

1. **Gate typing would be a desirable feature**, which would improve:

   readability, since it would no longer be necessary to parse a whole LOTOS description in order to guess what profiles can be accepted by a given gate.

modularity, since the interfaces of LOTOS processes (especially when considered as "black boxes") would be clearly defined.

reliability, since it would allow an early detection of certain classes of deadlocks. For instance, strange behaviours like "`G !false; ... || G !0; ...`" could be detected statically if gate `G` is declared to accept only boolean values. Also, it could allow the static detection of mistakes such as: omission of an experiment offer, supply of an experiment offer with a wrong sort, permutation of gate parameters in process instantiations, etc.

2. **A limited form of gate typing already exists in current** LOTOS. It is called "functionality" and is associated with the special gate "δ" used in "**exit**" and "**>> accept**" operators.

   Functionality declarations specify the acceptable profile for the "δ" gate: "**exit**" denotes profile $\langle\rangle$, "**exit** $(S_1, ..., S_n)$" denotes profile $\langle S_1, ..., S_n\rangle$, **noexit** means that there is no need to specify a profile since the "δ" gate will not be used at all.

   Functionality constraints restrict the use of the "**exit**" and "**>> accept**" operators according to functionality declarations. They attempt to prevent potential deadlocks on the "δ" gate (although a full deadlock prevention can not be obtained statically, since the problem is known to be undecidable).

   The functionality mechanism, although useful, is not so well integrated with the other LOTOS features, and often appears as a "special case".

   An appropriate gate typing mechanism for E-LOTOS should take into account the existing functionality mechanism.

3. **Polymorphic gates should be preserved**. A gate is said to be "polymorphic" (or "untyped") if it accepts any profile. In current LOTOS, all the gates are polymorphic. E-LOTOS should still allow polymorphic gates for at least three reasons :

   Upward compatibility: forbidding polymorphic gates would require all existing LOTOS specifications to be rewritten. Such a rewriting is not feasible practically, unless it is done automatically by some re-engineering tool.

   Design: during the early stages of an application design — and especially the architecture definition phase — design concepts are often not detailed enough: there is usually little knowledge about acceptable profiles. Polymorphic gates allow this problem to be deferred to further phases, still producing a valid LOTOS specification.

   Polyvalence: LOTOS is both a "calculus" and a "language". This is an essential feature of LOTOS. Researchers and teachers praise its conciseness and use it as a convenient set of notations to express concurrency and communication concepts. On the other hand, programmers are asking for more support in the development of large applications.

   An appropriate gate typing mechanism for E-LOTOS should conciliate both points of view.

   It is clear that untyped gates are not satisfactory for "programming in the large", and that typed gates should be used for this purpose.

   However, typed gates are not suitable for "programming in the small" since they introduce a syntactic overhead that is not justified when describing simple automata (for instance property observers) or experimenting with small descriptions.

4. **Gate overloading should be allowed**. A gate is said to be "overloaded" if it accepts a finite number of profiles (possible more than a single one). The essential difference between "overloaded" and "polymorphic" relies in the fact that the set of profiles accepted by an overloaded gate is enumeratively defined, whereas a polymorphic gate can accept any profile.

Requiring that a given gate can only accept a single profile is too restrictive for many applications (see, for instance, the example given in Annex A). Despite the fact that it is possible to write any LOTOS specification using only single profile gates, this approach is not always suitable: it increases the number of gates and goes against architectural principles. In particular, it should be possible to add a new profile to a gate in an existing specification without upsetting this specification by splitting this gate into two non-overloaded gates.

It is worth noticing that, in ESTELLE, channels may carry messages with different profiles.

5. **Gate typing should provide for structured events**. It is well-known that current LOTOS is not fully appropriate for large constraint-oriented descriptions. Such descriptions require so-called "structured events": it is not always necessary nor desirable that all constraining processes know about all experiment offers on a gate, since this is clearly a lack of structuring.

   This problem was pointed out by Pippo Scollo when developing OSI descriptions; he proposed shorthand notations for action denotations in order to have structured events [KS90]. This problem will be addressed in Section 4.

   A desirable gate typing mechanism should take this problem into consideration, by allowing action denotations to be structured.

6. **Gate type checking should be performed at compile-time**. In LOTOS and most high-level languages, type checking is done statically, at compile-time, not at run-time.

   The same principles should also apply to gate type checking: gate typing should only be a matter of static semantics and should bring no change in the existing dynamic semantics. In particular, gate typing should not introduce run-time overhead.

3 Proposal for a gate typing mechanism

3.1 Lexical changes

Three new keywords should be introduced in the syntax: "**channel**", "**endchan**", and "...".

Two new classes of identifiers should be introduced in the syntax:

- channel identifiers
- experiment identifiers

```
<channel-identifier> = <identifier> ;
<experiment-identifier> = <identifier> ;
```

3.2 Profiles

The syntax of a profile is defined as follows:

```
<experiment-declaration> ::=
       <sort-identifier>
   |   <experiment-identifier> ":" <sort-identifier> ;
<experiment-declaration-list> ::=
      <experiment-declaration>
   |  <experiment-declaration> "," <experiment-declaration-list> ;
<profile> ::=
```

```
          "(" ")"
       |  "(" <experiment-declaration-list> ")"
```

These are some examples of profiles:

- "`()`" denotes an empty profile, with no experiment offer.
- "`(bool)`" denotes a profile with a single experiment offer of sort `bool`.
- "`(bool, nat, bool)`" denotes a profile with three experiment offers, the respective sorts of which being `bool`, `nat` and `bool`.
- "`(E1:bool, E2:nat, E3:bool)`" denotes a profile with three experiment offers, the respective sorts of which being `bool`, `nat` and `bool` and the respective names of which being `E1`, `E2` and `E3`.
- "`(bool, E2:nat, bool)`" denotes a profile with three experiment offers, the respective sorts of which being `bool`, `nat` and `bool`, the name of the second experiment offer being `E2`, the first and third experiment offers being anonymous.

The experiment identifiers occurring in the same profile must be pairwise distinct.

Formally, a profile will be defined as an n-tuple $\langle E_1 : S_1, ..., E_n : S_n \rangle$ where $(n \geq 0)$, $S_1, ..., S_n$ are sort identifiers, and $E_1, ..., E_n$ are experiment identifiers. It is allowed to have some of the E_i undefined to represent anonymous experiment offers: by convention, anonymous E_i are supposed to be equal to a special value noted "$\perp$".

Two profiles $P' = \langle E'_1 : S'_1, ..., E'_m : S'_m \rangle$ and $P'' = \langle E''_1 : S''_1, ..., E''_n : S''_n \rangle$ are equal $(P' = P'')$ iff:

$$(m = n) \ \wedge \ (\forall i \in \{1, ..., m\}) \ \left((E'_i = E''_i) \ \wedge \ (S'_i = S''_i)\right)$$

3.3 Channel declarations

The syntax of LOTOS is extended with a notion of channel declaration. A channel declaration may appear in every place where a type declaration may appear. The corresponding syntax is:

```
<channel-declaration> ::=
   "channel" <channel-identifier> "is"
   <profile-list>
   "endchan" ;
<profile-list> ::=
      <profile>
   |  <profile> <profile-list>
```

These are some examples of channel declarations, from the simplest to the most complex ones:

```
channel C0 is
   ()
endchan

channel C1 is
   (bool)
endchan

channel C3 is
   ()
   (bool)
```

```
endchan

channel C4 is
   ()
   (bool)
   (nat, nat)
   (E1:bool, nat, E3:bool)
endchan
```

Intuitively, a channel is a gate type. Each profile in a channel definition specifies a permitted profile for all the gates typed with this channel. Channel definitions with more than a single profile allow gate overloading.

In channel definitions, sorts identifiers are visible with the same scope rules as in process definitions.

The occurrences of channel identifiers and experiment identifiers in the channel definitions are binding occurrences. These identifiers are visible in processes definitions with the same scope as sort identifiers.

The profiles occurring in the same channel definition are pairwise different.

It is not required that the experiment identifiers defined in the same channel definition be pairwise different. For instance, the following channel declaration is valid:

```
channel C5 is
   (bool)
   (E1:bool)
   (bool, nat)
   (E1:bool, nat)
   (bool, E1:nat)
endchan
```

The proposed syntax could be extended to allow a channel definition to import other channel definitions (as it is the case in LOTOS for type signatures). For instance, channels `C3` and `C4` above could be defined as:

```
channel C3 is C0, C1
endchan

channel C4 is C3
   (nat, nat)
   (E1:bool, nat, E3:bool)
endchan
```

3.4 Typed gate declarations

In LOTOS there are four occurrences of gate declarations: the "**hide**" operator, the "**choice**" operator, the "**par**" operator and process formal gate parameters.

The proposed mechanism extends the existing gate declaration syntax by allowing the gate identifier to be optionally followed by a channel identifier.

The proposed syntax is:

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

```
        <gate-identifier>
     |  <gate-identifier> "," <gate-identifier-list> ;
<gate-declaration> ::=
        <gate-identifier-list>
     |  <gate-identifier-list> ":" "any"
     |  <gate-identifier-list> ":" <channel-identifier> ;
<gate-declarations> ::=
        <gate-declaration>
     |  <gate-declaration> "," <gate-declarations> ;
<gate-selection> ::=
        <gate-declarations> "in" "[" <gate-identifier-list> "]"
<gate-selections> :=
        <gate-selection>
     |  <gate-selection> "," <gate-selections> ;
-- for the "hide" operator
        ... "hide" <gate-declarations> "in" ...
-- for the "choice" operator
        ... "choice" <gate-selections> "[]" ...
-- for the "par" operator
        ... "par" <gate-selections> <parallel-operator> ...
-- for process definitions
        ... "process" <process-identifier> "[" <gate-declarations> "]" ...
```

These are some example of possible declarations:

```
hide G0 in ...
hide G1, G2, G3 in ...
hide G0 : any in ...
hide G1 : any, G2, G3 : any in ...
hide G0 : C0 in ...
hide G1, G2 : C1, G3 : C2 in ...
```

The intuitive signification of gate declarations is the following:

- A gate not followed by "`:`" or followed by "`: any`" is an untyped (polymorphic) gate.
- A gate followed by "`:` C", where C is a channel identifier, is typed with this channel.

There is no syntactic ambiguity between "**any**" and a channel identifier, because the former is a reserved keyword.

There is a syntactic ambiguity regarding the respective precedences of "`,`" and "`:`". As a design choice, "`:`" is assigned a lower priority that "`,`". For instance, the following declaration:

```
hide G1, G2 : C1, G3, G4 : C2 in ...
```

will be parsed as:

```
hide {G1, G2} : C1, {G3, G4} : C2 in ...
```

and not as:

```
hide G1, {G2 : C1}, G3, {G4 : C2} in ...
```

Said differently, gate typing extends as much as possible to the left, in order to encourage strong typing. However, the second meaning can still be obtained using the "`any`" declaration:

```
hide G1 : any, G2 : C1, G3 : any, G4 : C2 in ...
```

3.5 Gate type equivalence

New statics semantics rules have to be introduced in order to ensure that "gate substitutions" are well-typed. There are three cases in LOTOS where a gate G'' is substituted to another gate G':

- **choice** G' **in** $[..., G'', ...]$
- **par** G' **in** $[..., G'', ...]$
- $P[..., G'', ...](...)$ **where process** $P[..., G', ...](...)...$

Substituting a gate G'' to a gate G' is only permitted if G' and G'' have a compatible gate type (see below).

Two gates G' and G'' have a compatible type (which is noted "$G' \equiv G''$") iff one of the following conditions is satisfied:

1. both G' and G'' are declared untyped (polymorphic);
2. G' and G'' are declared to be typed with the same channel identifier.

The definition above is based upon "name equivalence" for channels, instead of "structure equivalence". There are two reasons for this choice:

- Structure equivalence would lead to loose typing and unnecessary complexity in the static semantics. For instance, it would imply that the three channels below are equivalent because they contain the same profiles:

```
channel C1 is
   ()
   (bool)
   (nat, nat)
endchan

channel C2 is
   ()
   (bool)
   (nat, nat)
endchan

channel C3 is
   (nat, nat)
   ()
   (bool)
endchan
```

- Structure equivalence is not usual in LOTOS: two types with the same signature are not considered to be the same, two sorts with the same attached operations are not considered to be identical, etc.

Untyped gates are not compatible with typed gates, meaning that untyped gates are not "jokers" which can be used anywhere. This is a condition for strong typing. Otherwise, undesirable properties would ensue: for instance, given two typed gates G' and G'' and an untyped gate G, then $G \equiv G'$ and $G \equiv G''$ and — assuming that "$\equiv$" is an equivalence relation — by transitivity $G' \equiv G''$; consequently, "$\equiv$" would be the universal relation.

3.6 Tagged action denotations

The LOTOS syntax has to be extended for allowing experiment identifiers to be explicitly referenced in experiment offers. This is an essential point of the proposed mechanism, as far as structured events are concerned. Experiments identifiers play the role of "tags": they do not change the dynamic semantics of action denotations.

The current syntax of action denotations is the following:

```
<action-denotation> ::=
      <gate-identifier>
   |  <gate-identifier> <experiment-offer-list> <selection-predicate>
   |  "i" ;
<experiment-offer-list> ::=
      <experiment-offer>
   |  <experiment-offer-list> <experiment-offer> ;
<experiment-offer> ::=
     "?" <identifier-declaration>
   | "!" <value-expression> ;
```

The extended syntax requires to change only the definition of experiment offers:

```
<experiment-offer> ::=
     "?" <identifier-declaration>
   | "!" <value-expression>
   | <experiment-identifier> ":=" "?" <identifier-declaration>
   | <experiment-identifier> ":=" "!" <value-expression> ;
```

These are some examples of tagged action denotations:

```
channel C is
   ()
   (E1 : bool)
   (E2 : bool, E3 : nat, E4 : bool)
endchan

process P [G : C] : noexit :=
   G E1 := !true;
   G E1 := ?X:bool;
   G E2 := !true     E3 := !succ(0)  E4 := !false;
   G E2 := ?X:bool   E3 := ?Y:nat    E4 := ?Z:bool;
   G !true !succ(0) E4 := !false;
   stop
endproc
```

The experiment identifiers occurring in an tagged action denotation must be defined in the profile or the channel corresponding to the gate. Tagged action denotations can not be used if the gate is untyped gate.

3.7 Incomplete action denotations

It is also suitable to allow some experiment offers being omitted. For this purpose, a new symbol "..." is introduced in the action denotation syntax: it expresses the fact that the experiment offer list is incomplete. The syntax of these lists is modified as follows:

```
<experiment-offer-list> : :=
      <experiment-offer>
   |  "..."
   |  <experiment-offer> <experiment-offer-list> ;
```

Incomplete lists are to be completed with ?-offers containing "dummy" variables that will never be used. Incomplete lists are only allowed if no ambiguity can occur during completion (the completion algorithm will be detailed in section 3.8). For instance, with the channel definition of the previous examples, the following action denotations:

```
G E2 := !false ...;
G E3 := !succ (0) ...;
G E4 := ?Z:bool ...;
G E3 := !succ(0) E2 := !false ...;
G E2 := ?X:bool E3 := !succ(0) E4 := !true ...;
G !succ(0) ...;
stop
```

are expanded into standard LOTOS action denotations, where `p`, `q`, `r` are "dummy" variables that will never be used:

```
G !false   ?q:nat    ?r:bool;
G ?p:bool  !succ (0) ?r:bool;
G ?p:bool  ?q:nat    ?Z:bool;
G !false   !succ(0)  ?r:bool;
G ?X:bool  !succ(0)  !true;
G ?p:bool  !succ(0)  ?r:bool;
stop
```

If an action denotation is incomplete (i.e., if it contains the "`...`" symbol), its gate must not be untyped.

It is not required that all the experiment offers of an incomplete action be tagged.

If an action denotation is incomplete, the order of experiment offers is not significant. For instance, with the previous notations, the following code:

```
G E3 := !succ(0) E2 := !false ...;
G E4 := !false   E3 := !succ(0) ...;
G E4 := !false   E3 := !succ(0) E2 := !false ...;
stop
```

is expanded into:

```
G !false   !succ(0)   ?r:bool;
G ?p:bool  !succ(0)   !false;
G !false   !succ(0)   !false;
stop
```

In this proposal, order-free experiment offers are only allowed for incomplete actions. They are not permitted if the "`...`" symbol is absent.

3.8 Well-typed action denotations

The static semantics has to be extended with new rules, in order to deal with gate typing extensions. Basically, there are three tasks to be performed:

Action binding: it is necessary to check whether the action is compatible with the gate of the type. In case of gate overloading, it is necessary to select the appropriate profile, if any. This problem is close to the resolution of operation overloading, but the corresponding algorithm is much simpler since it only involves a limited form of pattern-matching.

Action completion: incomplete actions have to be completed, according to the appropriate profile.

Action reordering: incomplete actions have to be re-ordered, according to the appropriate profile.

Let A be an action denotation of the form:

$$G \quad [E_1\texttt{:=}]O_1...[E_n\texttt{:=}]O_n \quad [\ldots] \quad [\texttt{[}V\texttt{]}]$$

where:

- G is a gate;
- $O_1, ..., O_n$ is a possibly empty list ($n \geq 0$) of experiment offers;
- $E_1, ..., E_n$ are the experiment identifiers possibly attached to $O_1, ..., O_n$ (or "$\perp$" if not present);
- V is a selection predicate.

Let $S_1, ..., S_n$ be the respective sorts of the experiment offers $O_1, ..., O_n$.

Let Σ be the set of profiles defined as follows:

- If G is untyped then $\Sigma = \emptyset$.
- If G is typed with channel C then $\Sigma = \{P_1, ..., P_m\}$ where $(m \geq 1)$ and where $P_1, ..., P_m$ are the profiles contained in the definition of C.

To define if action A is well-typed, several cases are to be considered:

1. If $\Sigma = \emptyset$, then A is well-typed iff:
   - all the E_i $(i \in \{1, ..., n\})$ are equal to $\perp$, and
   - the "$\ldots$" symbol is absent
2. If $\Sigma \neq \emptyset$:
   (a) If the "$\ldots$" symbol is absent, then A is well-typed iff the cardinal of Σ' is equal to 1, where Σ' be the set of profiles defined as follows:
   $$\Sigma' = \{\langle E'_1 : S'_1, ..., E'_n : S'_n\rangle \in \Sigma \mid (\forall i \in \{1, ..., n\}) \ (E_i \in \{E'_i\} \cup \{\perp\}) \ \wedge \ (S'_i = S_i)\}$$
   (b) If the "$\ldots$" symbol is present, then A is well-typed iff the cardinal of Σ' is equal to 1, where Σ' be the set of profiles defined as follows:
   $$\Sigma' = \left\{\langle E'_1 : S'_1, ..., E'_m : S'_m\rangle \in \Sigma \mid \begin{array}{c} (n \leq m) \ \wedge \ (\exists \rho : \{1, ..., n\} \rightsquigarrow \{1, ..., m\} | \rho \ \textit{injective}) \\ (\forall i \in \{1, ..., n\}) \ (E_i \in \{E'_{\rho(i)}\} \cup \{\perp\}) \ \wedge \ (S_i = S'_{\rho(i)}) \end{array}\right\}$$

Σ' is the set of profiles that "match" the tagged experiment offers of A (possibly with reordering and completion, as figured out by function ρ). If the cardinal of Σ' is 0, then action A is not matched by any profile. If the cardinal of Σ' is greater than 1, then several matches are possible, which is considered to be an error (ambiguity).

The gate typing definition is "strong" in the sense that it does not allow any ambiguity. For instance, the following code will be rejected:

```
channel C is
   (STATUS:bool, REASON:nat)
   (STATUS:bool, CODE:nat)
   (STATUS:bool, REASON:nat, CODE:nat)
endchan
...
hide G:C in (G STATUS:=!true CODE:=?n:nat ... [n eq 0]; stop)
```

because two different profiles match the incomplete action denotation. Although such ambiguity might be useful (it introduces some kind on genericity) it is forbidden for safety reasons and to keep things simple.

3.9 Functionality declarations

The proposed mechanism allows to use a profile or a channel identifier for declaring the functionality of a process. The existing syntax:

```
<functionality-list> ::=
      ":" "noexit"
   |  ":" "exit"
   |  ":" "exit" "(" <sort-list> ")" ;
```

should be replaced by:

```
<functionality-list> ::=
      ":" "noexit"
   |  ":" "exit"
   |  ":" "exit" "(" <sort-list> ")" ;
   |  ""   -- the empty string
   |  ":" <channel-identifier> ;
```

A channel identifier occurring in a functionality list must only contain a single profile (possibly with experiment identifiers).

There is no syntactic ambiguity between "`noexit`" (*resp.* "**exit**") and the channel identifier, since "`noexit`" and "**exit**" are reserved keywords.

It is not clear at this point whether the existing constructs "**noexit**", "**exit**" and "**exit** $(S_1, ..., S_n)$" have to be dropped from LOTOS or kept for backward compatibility.

There would be great benefits in dropping them:

- "**noexit**" is too verbose and could be replaced by the empty string, introduced for this purpose;
- "**exit**" and "**exit** $(S_1, ..., S_n)$" introduce structure equivalence for profiles (see section 3.5 above). Therefore, gate type equivalence for the "δ" gate can not be the same as the equivalence defined in section 3.5 for ordinary gates. It must be extended in order to combine name equivalence (for channel identifiers) and structure equivalence (for functionalities defined with "**exit**").

  It is clear that "**exit**" and "**exit** $(S_1, ..., S_n)$" and the corresponding functionality rules are irregular with respect to the proposed gate typing mechanism. They could be replaced by channel names.

If these constructs are kept, the revised standard should strongly discourage their use, warn about their possible deprecation in a next revision of the standard, recommend the aforementioned replacement solutions, and advise compiler writers to flag the use of these constructs.

3.10 Extended exit and accept statements

Tagging, completion and reordering also apply to "**exit**" and "**accept**" statement, in the same way as for action denotations.

```
exit (E1 := false, succ (0), E3 := any bool)
exit (E2 := succ (0), ...)
exit (succ (0), ...)
exit (E3 := false, E1 := true, ...)
>> accept E1 := x:bool, y:nat, E3 := z:nat in
>> accept y:nat ... in
```

Missing argments in "**exit**" are replaced by "**any**" clauses. Missing arguments in "**accept**" are discarded.

3.11 Miscellaneous

To develop common vocabulary and notations amongst the LOTOS community, the standard library of E-LOTOS should contain a predefined channel identifier "**none**":

```
channel NONE is
   ()
endchan
```

which would allow to define gates without experiment offer:

```
hide G1:none, G2:any in ...
```

4 Assessment

In order to assess the benefits of gate typing, various existing LOTOS descriptions have been extended with gate typing informations. Due to a lack of space, it is not possible to include these descriptions in this paper, but such examples can be found, e.g., in [Gar94] which adapts the message router ("transit node") described in [Mou94], in [NN95] which features an X.25 switch, in [Man94a], etc.

From these experiments, it is clear that introducing gate typing strongly improves the readability and reliability of LOTOS descriptions.

Moreover, gate typing also provides an elegant solution to a well-known "structured events" problem which exists in LOTOS. This problem can be described as follows: generally, in real-life protocols, there are many experiment offers attached to a given gate; if several processes synchronize on this gate, it often happens that these processes only deal with a few experiment offers and need not know about the other ones. This is especially the case with "constraint-oriented" descriptions: by example, a process in charge of verifying that the recipients' addresses are correct should not bother with message contents. Using standard LOTOS, however, all experiment offers have to be explicitly mentioned in all processes.

To solve this problem, two main approaches have been proposed:

- The first approach is based on *compound events*, which require the introduction of a synchronous products of actions. The use of compound events has been advocated in [Mil85, Bri88], for instance.
- The second approach uses *shorthands* which allow, in a given process, to omit those experiment offers that are not relevant to the behaviour of the process. Shorthands have been used in large specifications, such as the formal description in LOTOS of the Transport Protocol [KS90].

The example shown below is precisely borrowed from [KS90]. In this example, gate `p` has 7 experiment offers and is used by many small processes. Each process only works on a few experiment offers and needs not know about the other ones. Due to the use of shorthands, these processes are not written in standard LOTOS. Shorthands are similar to incomplete actions: they allow to omit experiments, which are to be replaced by `?`-offers with dummy variables. To work properly, shorthands assume that each gate has only a single profile and that the sorts of all experiments offers in this profile are pairwise distinct. Under this assumption, there is no need for experiment identifiers:

```
(*
The events at gate p have the following structure:
p ?tr:TPId ?ni:NCId ?cl:Class ?d:Dir ?c:Copy ?tpdu:ETPDU ?err:TPErr
*)

process TCLocalRef [p] (lr:TPId) : noexit:=
   p !lr ;
   TCLocalRef [p] (lr)
endproc

process FreeReference [p, r] (lr:Ref) : noexit:=
   p ?tr:TPId [Qual(tr) eq Local and (Ref(tr) eq lr)] ;
   BoundReference [p, r] (Local(lr))
endproc

process UseBoundReference [p] (tr:TPId) : noexit :=
   p !tr ?d:Dir ?c:Copy ?tpdu:ETPDU [not(AssignLocalRef(d, c, tpdu))] ;
   UseBoundReference [p] (tr)
endproc

process TCIdByRef [p] : noexit:=
   p ?tr:TPId !Send !New ?tpdu:ETPDU [LocalSrcRef(tr, tpdu)] ;
   TCLocalRef [p] (Local(Ref(tr)))
   []
   p ?tr:TPId !Recv ?tpdu:ETPDU [RemoteSrcRef(tr, tpdu)] ;
   TCRemoteRef [p] (tr)
endproc
```

The same example can be written in a very similar way using the proposed gate typing mechanism. Basically, it is sufficient to define a channel with a single profile (which formalizes the comments contained in the original description) and to add "`...`" symbols to incomplete action denotations. Also, to improve readability, `!`-offers have been tagged, but this is not mandatory:

```
channel TPDU_transfer is
   (tr:TPId, ni:NCId, cl:Class, d:Dir, c:Copy, tpdu:ETPDU, err:TPErr)
endchan

process TCLocalRef [p:TPDU_transfer] (lr:TPId) : noexit:=
   p tr:=!lr ... ;
   TCLocalRef [p] (lr)
endproc

process FreeReference [p:TPDU_transfer, r] (lr:Ref) : noexit:=
   p ?tr:TPId ... [Qual(tr) eq Local and (Ref(tr) eq lr)] ;
   BoundReference [p, r] (Local(lr))
endproc
```

```
process UseBoundReference [p:TPDU_transfer] (tr:TPId) : noexit :=
   p tr:=!tr ?d:Dir ?c:Copy ?tpdu:ETPDU ... [not(AssignLocalRef(d, c, tpdu))] ;
   UseBoundReference [p] (tr)
endproc

process TCIdByRef [p:TPDU_transfer] : noexit:=
   p ?tr:TPId d:=!Send c:=!New ?tpdu:ETPDU ... [LocalSrcRef(tr, tpdu)] ;
   TCLocalRef [p] (Local(Ref(tr)))
   []
   p ?tr:TPId d:=!Recv ?tpdu:ETPDU ... [RemoteSrcRef(tr, tpdu)] ;
   TCRemoteRef [p] (tr)
endproc
```

It is worth mentioning that gate typing is more general than shorthands, since it does not preclude the experiments offers in the profile from having the same sorts.

There are essential differences between the "compound events" approach and the "gate typing" approach proposed here:

- Gate typing is performed statically (at compile-time) whereas compound events are more likely computed at run-time;
- Introducing gate typing does not modify the existing dynamic semantics of LOTOS, whereas compound events require major changes;
- Gate typing is based on fairly standard type-checking algorithms, which could easily be added to existing tools. Conversely, there is little experience about effective implementation of compound events.
- Compound events do not remove the need for gate typing. With compound events, a gate typing mechanism is still necessary: it must be even more complex than the current proposal, in order to deal with action products.

It would be interesting to decide whether compound events are more expressive than gate typing and whether this additional expressiveness, if any, is worth the additional complexity in the static and dynamic semantics.

5 Conclusion

A gate typing mechanism for LOTOS has been proposed, which meets the requirements for readability, modularity, and reliability, exposed in section 2. It also provides a simple and elegant solution for the need of structured events. It is totally upward compatible with current LOTOS, in the sense that any existing LOTOS program would remain a valid program under the proposed extension.

In the framework of E-LOTOS standardization process, the first proposal for typing LOTOS gates was made by José Manas [Man94b]. Then, the proposal described in this paper was input [Gar94], which introduced the concepts of experiment identifiers, tagged action denotations, and incomplete action denotations, with an emphasis on upward compatibility, gate overloading, and gate polymorphism. The initial proposal [Man94b] as gradually evolved [Man94c, Man94a] so that both proposals are presently very close.

Possible extensions to the proposed gate typing mechanism could be the introduction of sub-typing (as in [Man94a]) and/or structure equivalence (as in [NN95]), but it is still an issue to decide whether the supposed benefits of these extensions are worth their complexity.

Acknowledgements

The author is grateful to Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Abdelbarim Nimour for their helpful comments and suggestions.

References

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

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

[ISO88] 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.

[Jua93] Juan Quemada, editor. Initial Draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG1 N8023 Project 1.21 Q1/48.6, November 1993.

[Jua94a] Juan Quemada, editor. Revised Draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG1 N1314 New Work Item Q48.6, March 1994.

[Jua94b] Juan Quemada, editor. Working Draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG1 N1349 Project 1.21.20.2.3, December 1994.

[KS90] Harro Kremer and Giuseppe Scollo. Formal description in LOTOS of the OSI transport protocol defined in ISO/IS 8073. 1990.

[Man94a] José A. Manas. Typed Gates. Annex C of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS, December 1994.

[Man94b] José A. Manas. Typed Gates. Contribution MAD7 to the Madrid E-LOTOS meeting, ISO/IEC JTC1/SC21/WG1/N2802, January 1994.

[Man94c] José A. Manas. Typed Gates. Annex C of ISO/IEC JTC1/SC21/WG1 N1314 Revised Draft on Enhancements to LOTOS, March 1994.

[Mil85] G. J. Milne. CIRCAL and the Representation of Communication, Concurrency, and Time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[Mou94] Laurent Mounier. A LOTOS Specification of a Transit-Node. Rapport SPECTRE 94-8, VERIMAG, Grenoble, March 1994.

[NN95] Elie Najm and Abdelbarim Nimour. Extending Gate Typing to Mobile LOTOS. ISO/IEC JTC1/SC21/WG1 N1358, February 1995.