French-Romanian Integrated Proposal for the User Language of E-LOTOS

Version 1.0 (Draft)

Input document of ISO/IEC JTC1/SC21/WG7/1.21.20.2.3

'Enhancements to LOTOS'

Kansas City meeting, May 1996

Hubert Garavel and Mihaela Sighireanu*
INRIA Rhône-Alpes
VERIMAG — Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE

Abstract

This document attempts to merge several existing proposals for E-LOTOS, in order to provide a unified definition for the User Language of E-LOTOS, covering several aspects that were so far addressed independently each from the others. This proposal covers the data type part of E-LOTOS, as well as the behaviour part, including gate typing.

The document first provides rationale for the proposed language features. Then, an abstract syntax of the User Language is given. The static and dynamic semantics of this User Language are defined formally, by using successive translations of the User Language into refined, more primitive sub-languages. Finally, several use examples of the User Language are given, including the LOTOS standard data type library and some of the type definitions contained in the so-called LOTOSPHERE proposal.

*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".

Contents

1 Introduction **5**

2 Rationale for the proposed enhancements in the behaviour part **5**

3 Rationale for the proposed enhancements in the behaviour part **6**
3.1 Giving a printable name to the "δ" gate 6
3.2 Turning the specification identifier into an ordinary process identifier 6
3.3 Turning the reserved keyword "**i**" into a predefined gate identifier 7
3.4 Introducing two "**case**" operators 8
3.5 Introducing an "**if**" operator 11
3.6 Extending the "**let**" operator 12
3.7 Introducing a "**rename**" operator 12
3.8 Removing the "**choice**" and "**par**" operators over gate lists 17
3.9 Using a bracketed syntax 17
3.10 Introducing a "**par**" operator on finite value domains 20
3.11 Allowing arbitrary synchronizations 20
3.12 Introducing "n among m" synchronization 24
3.13 Introducing exceptions in the behaviour part 26
3.14 Unifying the ";" and ">>" operators 32
3.15 Introducing iterators in the behaviour part 40
3.16 Removing the "**where**" clause from process definitions 41
3.17 Simplifying process definitions 42
3.18 Abbreviating gate parameters lists 44
3.19 Abbreviating value parameters lists 47

4 Definition of the user language **49**
4.1 Schema overview 49
4.2 Notations 50
4.3 Abstract syntax of the full user language 51
4.3.1 Declarations 51
4.3.2 Patterns 52
4.3.3 Match expressions 52
4.3.4 Value expressions 52
4.3.5 Behaviour expressions 54
4.4 Abstract syntax of the reduced user language 58
4.4.1 Declarations 58
4.4.2 Patterns 59
4.4.3 Match expressions 59
4.4.4 Value expressions 59
4.4.5 Behaviour expressions 60
4.5 Abstract syntax of the minimal core language 62
4.5.1 Declarations 62
4.5.2 Patterns 62
4.5.3 Match expressions 62
4.5.4 Values terms 63
4.5.5 Behaviour expressions 63
4.6 Translation function from full to the reduced user language 65
4.6.1 Contexts 65

4.6.2 Translation function 68
4.6.3 Translation of declarations 68
4.6.4 Translation of patterns 69
4.6.5 Translation of match expressions 70
4.6.6 Translation of value expressions 70
4.6.7 Translation of behaviour expressions 73
4.7 Static semantics of the reduced user language 79
4.7.1 Notations 79
4.7.2 Semantical objects 79
4.7.3 Static semantics of variable lists 81
4.7.4 Static semantics of gate lists 82
4.7.5 Static semantics of declarations 82
4.7.6 Static semantics of patterns 84
4.7.7 Static semantics of match expressions 84
4.7.8 Static semantics of value expressions 85
4.7.9 Static semantics of behaviour expressions 87
4.8 Translation function from reduced user language to minimal core language 89
4.8.1 Translation of declarations 90
4.8.2 Translation of patterns 91
4.8.3 Translation of match expressions 91
4.8.4 Translation of value expressions 91
4.8.5 Translation of behaviour expressions 94
4.9 Dynamic semantics of the minimal core language 97
4.9.1 Environments 97
4.9.2 Dynamic semantics of variable list 97
4.9.3 Dynamic semantics of declarations 97
4.9.4 Dynamic semantics of patterns 97
4.9.5 Dynamic semantics of match expressions 98
4.9.6 Dynamic semantics of expressions 99
4.9.7 Dynamic semantics of behaviours 100

5 Conclusion **103**

A Expressing the "if" operator in standard LOTOS **103**

B Sequential composition and bracketed syntax in LOTCAL **107**

C A simplified transport service **109**

D A simplified sliding window protocol **112**

E Definition of LOTOS standard data types (IS 8007) in E-LOTOS **116**
E.1 Boolean 116
E.2 FBoolean 118
E.3 Element 118
E.4 Set 119
E.5 BasicNonEmptyString 122
E.6 RicherNonEmptyString 122
E.7 NonEmptyString 124
E.8 String 125

E.9 BasicNaturalNumber . . . 127
E.10 NaturalNumber . . . 128
E.11 NatRepresentations . . . 130
E.12 HexNatRepr . . . 130
E.13 HexString . . . 130
E.14 HexDigit . . . 131
E.15 DecNatRepr . . . 133
E.16 DecString . . . 134
E.17 DecDigit . . . 134
E.18 OctNatRepr . . . 136
E.19 OctString . . . 137
E.20 OctDigit . . . 137
E.21 BitNatRepr . . . 139
E.22 BitString . . . 140
E.23 Bit . . . 140
E.24 Octet . . . 142
E.25 OctetString . . . 144

F Translation of LOTOSPHERE data types in E-LOTOS **145**
F.1 BOOL . . . 145
F.2 NAT . . . 145
F.3 INT . . . 146
F.4 FRAC . . . 147
F.5 FLOAT . . . 150
F.6 CHAR . . . 151
F.7 STRING . . . 152

G History of this document **153**
G.1 Previous proposals regarding the data part . . . 153
G.2 Previous proposals regarding the behaviour part . . . 153
G.3 Previous proposals regarding the gate typing . . . 154
G.4 History of this document . . . 154

1 Introduction

This document attempts to merge several existing proposals for E-Lotos that were so far considered as separate parts in the latest version of the E-Lotos Revised Working Draft [Que96b]. These proposals are the following:

- Annex A Part 1 of [Que96b] ("A Proposal for the Data Type Part of E-LOTOS Applicable to the Formal Description of OSI and ODP Standards").
- Annex C Proposal 2 of [Que96b] ("On the Introduction of Gate Typing in E-LOTOS"). However, some ideas of Annex C Proposal 1 ("Tagged/Typed Gates") are also taken into account, for instance the restrictions concerning the mix of tagged and positional gates.
- Annex F Part 1 of [Que96b] ("Generalized Termination, Enabling and Disabling") as the exception mechanism ("**trap**" operator) proposed in this document seems to be a generalization of the generalized enabling operator ">*t*>".
- Annex F Part 2 of [Que96b] ("A Wish List for the Behaviour Part of LOTOS").

This document benefits from the discussions with Alan Jeffrey, Guy Leduc, Charles Pecheur, and Ricardo Pena during a short-term scientific mission (April, 22–26, 1996) supported by the European Commission under Cost 247 Action and European-Canadian project Eucalyptus-2.

A detailed history of the successive versions of this document can be found in Annex G.

Due to the short lapse of time between the Liège scientific mission and the Kansas City meeting, the document is not complete yet and some of its sections are not fully up to date. It is organized as follows.

Section 2 presents the underlying motivations for the data part of the user language. This section is still to be provided.

Section 3 presents the underlying motivations for the data part of the behaviour language. This section is identical to Section 2 of [Que96b, Annex F Part 2]. It should be updated since it is not fully up to date with the latest definition of the User Language given in Section 4.

Section 4 provides a detailed proposal for the User Language. According to the discussions of the Liège meeting (December 1996), we distinguish between two levels of language: a *Core Language*, with a minimal set of operators, and a *User Language*, with convenient notations which can be expanded into the core language.

Use examples of the User Language can be found in Annexes C to F of this document. In particular, Annex E shows how Lotos standard type library could be expressed in E-Lotos, and Annex F does the same with some of the built-in types of the so-called "Lotosphere proposal".

2 Rationale for the proposed enhancements in the behaviour part

To be provided.

3 Rationale for the proposed enhancements in the behaviour part

In this Section, we propose a collection of enhancements to the behaviour part of LOTOS. These enhancements aim at solving several problems found in LOTOS and making the behaviour part of E-LOTOS more expressive, simpler, symmetric with the data part of E-LOTOS, and compatible with usual programming languages. In this Section, we explain informally the motivations for the proposed enhancements (a formal presentation will be given in Section 4). The enhancements are presented separately in different sub-sections. Therefore, each sub-section can be read independently from the other ones. However, if one enhancement relies upon another one, the dependence is explicitly stated.

3.1 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.

3.2 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 change: the specification identifier should be a process identifier and should be visible in the behaviour expression following the "**behaviour**" (or "**behavior**") keyword.

The proposed change is fully upward compatible.

3.3 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.

3.4 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.

[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

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:

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

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

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:

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

  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 symmetrical 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. This is the solution that we prefer.

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

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

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

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:

$$
\begin{array}{l}
\textbf{case} \\
E::X{:}T \rightarrow \\
\quad \textbf{case} \\
\quad X::P_0^0 \textbf{ or } \ldots \textbf{ or } X::P_{m_0}^0 \ [\textbf{where } E_0] \rightarrow B_0 \\
\quad \ldots \\
\quad X::P_0^n \textbf{ or } \ldots \textbf{ or } X::P_{m_n}^n \ [\textbf{where } E_n] \rightarrow B_n \\
\quad [\textbf{otherwise } B_{n+1}] \\
\quad \textbf{endcase} \\
\textbf{endcase}
\end{array}
$$

The above translation is designed to evaluate expression E only once, by introducing a variable X whose type T is the type of E. It is to be noticed that, if the evaluation of E raises an exception, then the expanded form also raises an exception.

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

$$
\begin{array}{l}
\textbf{case } E \textbf{ in} \\
true \rightarrow B \\
false \rightarrow \textbf{stop} \\
\textbf{endcase}
\end{array}
$$

□

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.

3.5 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:

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

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

> 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.

3.6 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 `::` P_0 **and** ... **and** E_n `::` $P_n \rightarrow B_0$
  endcase

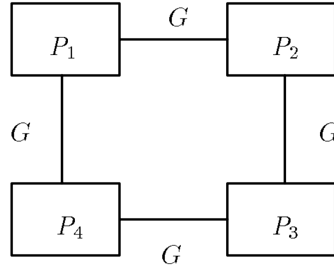
This modification is "almost" upward compatible with existing LOTOS because E-LOTOS patterns include variable declarations "$V : T$" as a particular form of patterns. To translate a LOTOS "**let**" operator into an E-LOTOS "**let**" operator, two changes have to be performed:

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. This is to ensure homogeneous notations between "**let**" operators, "**rename**" operators (see Section 3.7), process instantiations (see Sections 3.18 and 3.19), and function calls (see the data type language).

3.7 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 two auxiliary gates G' and G'', which are later renamed in G, the ring can be described as follows:

$$
\begin{array}{l}
\textbf{rename } G' := G, G'' := G \textbf{ in} \\
\quad (\\
\quad\quad (P_1[G'', G'] \; |[G']| \; P_2[G', G'']) \\
\quad |[G'']| \quad\quad (P_3[G'', G'] \; |[G']| \; P_4[G', G'']) \\
\quad)
\end{array}
$$

The description of such complex synchronization patterns will also be discussed in Sections 3.11 and 3.12.

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". 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/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:

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

     but has to write instead:

     $P[G'_0, ..., G'_n, \mathcal{G}](\mathcal{E})$
     where
     process $P[G_0, ..., G_n, \mathcal{G}](\mathcal{V}) : \mathcal{F}$ `:=`
     B_0
     endproc

     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:

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

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:

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

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

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

- There are additional static semantic constraints.
  - For each renaming clause R_i, the names of the variables $V_i^1, ..., V_i^{p_i}$ should be pairwise distinct.
  - 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 $(1 \leq j \leq p_i)$ 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:

$$\begin{array}{l}(\forall i, j \in \{0, ..., n\})\;\; i \neq j \Longrightarrow \\ (G_i \neq G_j) \;\vee\; ((\textit{both } R_i \textit{ and } R_j \textit{ have a } (...) \textit{ clause}) \;\wedge\; (p_i \neq p_j) \;\vee\; (\exists k \in \{1, ..., p_i\} \mid T_i^k \neq T_j^k))\end{array}$$

- 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 $T_i^1, ..., T_i^{p_i}$ 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. □

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 (*header*, V, *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:

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

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:

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

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.

However, the pre-renaming semantics does not solve the issue of complex synchronizations mentioned at the beginning of this Section. □

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

3.8 Removing the "choice" and "par" operators over 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 3.7 above). For instance:

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

  could be expressed as:

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

  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.

3.9 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.

However, this scheme proves to be difficult to many users. For instance, the behaviour expression "B_1 >> B_2 [] B_3" is not parsed as "(B_1 >> B_2) [] B_3" but as "B_1 >> (B_2 [] 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.

[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.

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 for his LOTCAL language. This solution is worth to be considered for the design of E-LOTOS. In LOTCAL, 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. All these forms share a common syntax:

  par sync *synchronization_clause*
  B_0
  `||`
  ...
  `||`
  B_n
  endpar

  Generalized parallel composition operators will be further discussed in Sections 3.11 and 3.12.

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

```
seq
   A0
;
   ...
;
   An
;
   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 3.14 below), some A_i can also be replaced by behaviour expressions.

- 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 synchronization_clause
   B
endpar
```

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

Remark
According to the proposals made in Section 3.8, the iterations over gate lists should not be retained for E-LOTOS. □

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

Remark
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. □

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

3.10 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.:

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

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 "$\mathit{is_finite}(T)$" 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.

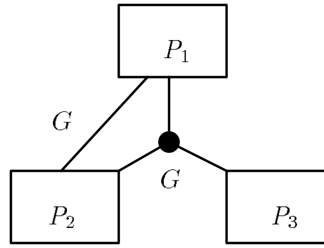
3.11 Allowing arbitrary synchronizations

We believe that the parallel composition operators ('`||`", "`|||`", and "`|[...]|`") which exist in LOTOS are not fully satisfactory, for several reasons:

1. The main advantage of these operators relies in the fact that they are simple from an algebraical point of view: they are binary, commutative, and associative.

2. However, from the specifier point of view, these operators are not very intuitive. They are probably too far from the usual vision of a system seen as a network of parallel processes connected by (multiway) communication links.

3. It is known that LOTOS parallel operators are not powerful enough to describe all possible of networks of communicating processes. The network given in Section ?? is an example, although it can be described using both parallel composition and renaming operators. Another example is the following:



4. However, even when it is possible to describe a network using LOTOS parallel operators, the solution is not always easy:
   - Experience indicates that it proves to be difficult for LOTOS beginners.
   - This is also difficult for software tools which attempt to translate a network (represented graphically, e.g., as an SDL view) into a LOTOS behaviour expression: this problem is equivalent to a system of boolean equations. To solve this problem efficiently, the use of Binary Decision Diagrams is advised.
   - There are often many possible solutions, all of them looking very different.
   - The resulting behaviour expression is not always readable, especially if it involves renaming (through process instantiations).

To solve these problems, we would like to introduce in E-LOTOS a generalized parallel operator which allows to express directly networks of communicating processes. This would be helpful to write and read E-LOTOS descriptions and to develop tools that translate graphical views into behaviour expressions and vice-versa.

First, in order to gain expressiveness and convenience, it is obvious that we have to shift from LOTOS' binary parallel operators to n-ary operators, as suggested by Ed Brinksma (see Section 3.9).

Second, we propose to introduce in E-LOTOS the notion of *synchronization product* (or *synchronization vector*) found in languages such as MEC [Arn92] or AUTO/AUTOGRAPH [RS90].

The general parallel operator has the following syntax:

par $G_1 : \Sigma_1, ..., G_m : \Sigma_m$ **in**
   B_0
||
   ...
||
   B_n
endpar

where $G_1, ..., G_m$ are gates and where $\Sigma_1, ..., \Sigma_m$ are *non-empty* subsets of $\{0, ..., n\}$. To allow different synchronizations on the same gate, we do not require that all gates $G_1, ..., G_m$ are pairwise distinct. All Σ_i's must be computable statically (i.e., at compile-time); in a first approximation, we can assume that, syntactically, all Σ_i's are merely lists of integer constants.

Informally, the semantics of this operator is the following. The behaviour expressions $B_0, ..., B_n$ execute concurrently. The intuitive semantics of this operator is that, for each couple (G_i, Σ_i), all behaviour expressions B_j such that j belongs to Σ_i synchronize on gate G_i. Therefore, the set Σ_i associated to a gate G_i is the set of all indexes j such that B_j synchronize on gate G_i. As in LOTOS, the termination is synchronous: all B_i's synchronize on the "δ" gate.

Remark
If there are several subsets Σ_i of behaviour expressions B_j ready to synchronize together, a non-deterministic choice will be made between the different possible subsets Σ_i. □

Remark
For compatibility with the semantics of LOTOS and to avoid the problem of large, unreadable synchronized products, we only allow products of the form "$G \times ... \times G \rightarrow G$" (meaning that gates only synchronize if they have the same name) whereas the general synchronization product has the form "$G_1 \times ... \times G_n \rightarrow G$" (meaning that gates $G_1, ..., G_n$ can synchronize leading to an event labelled gate G). □

Remark
We have selected this parallel operator for the core behaviour language because of its expressiveness. However, we acknowledge that it is not particularly user-friendly: if a behaviour expression B_i is added or removed, or if the behaviour expressions B_i are permuted, the the subsets S_i must be updated accordingly. It would be perhaps suitable to use symbolic identifiers (*tags*) instead of index numbers. This is left for further study. □

For instance, using this operator, the example networks of Sections ?? and 3.11 can be described as:

par $G : \{1, 2\}, G : \{2, 3\}, G : \{3, 4\}, G : \{4, 1\}$ **in**
B_1 || B_2 || B_3 || B_4
endpar

and:

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

More generally, this parallel composition operator allows to obtain, as particular cases, the parallel operators proposed by Ed Brinksma in his LOTCAL calculus [Bri88]. These n-ary operators generalize the binary operators that exist in standard LOTOS. Brinksma proposes four kinds of operators:

1. The first operator performs synchronization on gates $G_0, ..., G_m$. It is noted:

   par sync $G_0, ..., G_m$ **in**
   B_0 || ... || B_n
   endpar

   and is equivalent to:

   par $G_1 : \{0, ..., n\}, ..., G_m : \{0, ..., n\}$ **in**
   B_0 || ... || B_n
   endpar

   It is also equivalent to (B_0 |[$G_0, ..., G_m$]| ... |[$G_0, ..., G_m$]| B_n).

2. The second operator performs synchronization on all visible gates. It is noted:

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

and is equivalent to:

$$\begin{array}{l}\textbf{par } G_1 : \{0, ..., n\}, ..., G_m : \{0, ..., n\} \textbf{ in}\\ \quad B_0 \ \texttt{||} \ ... \ \texttt{||} \ B_n\\ \textbf{endpar}\end{array}$$

where $G_1, ..., G_m$ denote all visible gates in $B_0, ..., B_m$. It is also equivalent to $(B_0 \ \texttt{||} \ ... \ \texttt{||} \ B_n)$.

3. The third operator performs full interleaving. It is noted:

$$\begin{array}{l}\textbf{par } [\textbf{sync none}]\\ \quad B_0 \ \texttt{||} \ ... \ \texttt{||} \ B_n\\ \textbf{endpar}\end{array}$$

and is equivalent to:

$$\begin{array}{l}\textbf{par in}\\ \quad B_0 \ \texttt{||} \ ... \ \texttt{||} \ B_n\\ \textbf{endpar}\end{array}$$

It is also equivalent to $(B_0 \ \texttt{|||} \ ... \ \texttt{|||} \ B_n)$.

4. The fourth operator proposed by Ed Brinksma is derived from CSP parallel composition and performs synchronization on the common gates of all processes. It is noted:

$$\begin{array}{l}\textbf{par sync common}\\ \quad B_0 \ \texttt{||} \ ... \ \texttt{||} \ B_n\\ \textbf{endpar}\end{array}$$

Unfortunately, this operator is highly context-dependent; in particular, it is not a congruence with respect to strong equivalence. If we note *op* the binary form of the parallel composition operator proposed by Ed Brinksma, the behaviour expression $B_1 \ op \ B_2$ depends upon $\mathcal{L}(B_1) \cap \mathcal{L}(B_2)$, where $\mathcal{L}(B)$ denotes the set of gates visible in B. For instance, let $B_1 = \textbf{stop}$, let $B_2 = \textbf{stop} \ \texttt{||} \ G \ ; \ \textbf{stop}$ and let $B_3 = G \ ; \ \textbf{stop}$. B_1 and B_2 are strongly equivalent, but $B_1 \ op \ B_3$ and $B_2 \ op \ B_3$ are not. Indeed:

$$B_1 \ op \ B_3 = G \ ; \ \textbf{stop}$$

because there is no synchronization on gate G since $\mathcal{L}(B_1) = \emptyset$. Conversely:

$$B_2 \ op \ B_3 = \textbf{stop}$$

since synchronization on gate G is mandatory as $\mathcal{L}(B_2) = \mathcal{L}(B_3) = \{G\}$.

Therefore, we propose here a modified version of Brinksma's operator, which preserves congruence. The syntax is:

$$\begin{array}{l}\textbf{par sync common}\\ \quad \mathcal{G}_0 : B_0\\ \texttt{||}\\ \quad \dots\\ \texttt{||}\\ \quad \mathcal{G}_n : B_n\\ \textbf{endpar}\end{array}$$

where $\mathcal{G}_0, ..., \mathcal{G}_n$ are gate lists. The intuitive semantics is the following: for each gate G in $\mathcal{G}_0 \cup ... \cup \mathcal{G}_n$, all behaviour expressions B_i such that $G \in \mathcal{G}_i$ have to synchronize on gate G. This operator is equivalent to:

```
par G1 : Σ1, ..., Gm : Σm in
   B0
||
   ...
||
   Bn
endpar
```

where $\{G_1, ..., G_m\} = \mathcal{G}_0 \cup ... \cup \mathcal{G}_n$ and where, for each $i \in \{1, ..., m\}$, we have $\Sigma_i = \{j \mid (0 \leq i \leq m) \wedge (G_i \in \mathcal{G}_j)\}$.

Remark
The "**par sync common**" is less expressive than the general "**par**" operator proposed at the beginning of this section. In particular, it does not allow different synchronizations groups on the same gate. □

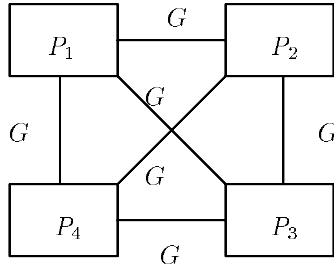
Remark
Regarding expressiveness, our proposal is upward compatible with LOTOS since LOTOS binary parallel operators can still be obtained as particular cases of n-ary parallel operators. □

Remark
The proposed synchronization scheme should be adapted to the "**par**" operator on value domains described in Section 3.10. □

3.12 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, a fully connected 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 either binary rendez-vous or a binding object.

- "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.

The above network can be described using the generalized parallel operator proposed in Section 3.11:

par sync $G : \{1,2\}, G : \{1,3\}, G : \{1,4\}, G : \{2,3\}, G : \{2,4\}, G : \{3,4\}$ **in**
  B_1 || B_2 || B_3 || B_4
endpar

However, this notation is not very convenient, since it requires to write all subsets of $\{1, ..., 4\}$ the cardinal of which is 2.

We therefore propose an abbreviation to solve this problem. Let's consider the generalized parallel operator:

par $G_1 : \Sigma_1, ..., G_m : \Sigma_m$ **in**
  B_0
||
  ...
||
  B_n
endpar

In the list of clauses between the keywords "**par**" and "**in**", we also accept clauses of the form "G # p", where $1 \leq p \leq n+1$. These clauses are merely shorthand notations. Each clause "G # p" has to be expanded into a list of clauses:

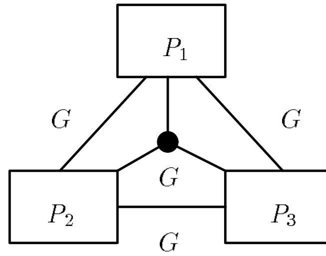
$$G : \Sigma_1, G : \Sigma_2, ..., G : \Sigma_{C^p_{n+1}}$$

where $\Sigma_1, \Sigma_2, ...\Sigma_{C^p_{n+1}}$ are all subsets of $\{0, ..., n\}$ the cardinal of which is p. There are C^p_{n+1} such subsets, where C^b_a denotes the binomial coefficients.

Remark
A clause of the form "G # p" with $p = 1$ means that gate G is asynchronous for all B_i's. Ommitting this clause would give the same result.

A clause of the form "G # p" with $p = n+1$ means that gate G is synchronous for all B_i's. This clause is equivalent to "$G : \{0, ..., n\}$". □

Remark
We allow different degrees of synchronization on the same gate. For instance, the following network:



can be obtained as follows:

$$\begin{array}{l}\textbf{par sync } G \;\#\; 2, G \;\#\; 3 \textbf{ in}\\ \quad B_1 \;||\; B_2 \;||\; B_3\\ \textbf{endpar}\end{array}$$

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. □

Remark
The proposed "n among m" synchronization scheme should be adapted to the "**par**" operator on value domains described in Section 3.10. □

3.13 Introducing exceptions in the behaviour part

The need for introducing exceptions (also called generalized termination and enabling) in E-Lotos has been clearly identified in [QA92] and [Que96a, Annex F]. There have been also several proposals for introducing exceptions as a fundamental concept in concurrent languages, e.g. Atp (Algebra of Timed Processes) [NS90, NS94] and Esterel [Ber93].

As regards the data part of E-Lotos, exceptions are currently part of the existing proposals, including [JGL+95] and [SG95]. We propose here an exception mechanism for the behaviour part, compatible with the one given in [SG95].

In our approach, exceptions constitute a new class of identifiers: we will note them $X, X', X'', X_1, X_2, \ldots$. There exist some predefined exceptions which need not be declared explicitly, e.g., the "no_match" exception that indicates a missing clause is a "**case**" operator.

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

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

Remark
Actually, the arguments of the "**raise**" operator should not be mere value expressions $E_1, ..., E_n$. To ensure symmetry with the gate typing and abbreviated process instantiations (see Sections 3.18 and 3.19), one should also allow "$V_i := E_i$" clauses (where V_i is the name of the i-th formal argument of exception X), "**any** T_i" clauses (as in the "**exit**" operator of LOTOS) and ellipsis clauses "**...**". The dynamic semantics of the two latter constructs would be that of a non-deterministically chosen value. □

Remark
Like ML and unlike ADA, our exceptions carry typed values. We believe that it would be appropriate to introduce a notion of "exception typing", similar to "gate typing", which would allow to declare exceptions together with the types of their parameters. However, we will not consider this issue here. □

As regards the "**trap**" operator, we will call B_0 the *normal behaviour* and $X_i ... \rightarrow B_i$ the *exception handlers*.

The "**trap**" operator declares the exception identifiers X_i and the variable identifiers V_i^j. The exceptions X_i are only visible in B_0 and the variables V_i^j are only visible in B_i (and in the "**raise** X_i" operators occurring in B_0). There should be static semantics constraints ensuring that an exception X is always raised with a list of values $E_1, ..., E_n$ compatible, in number and types, with the list of variables $V_1 : T_1, ..., V_n : T_n$ attached to the definition of X.

Remark
Many languages (e.g., ADA, ATP, ESTEREL and ML) place the exception handlers after the normal behaviour, because they lay the emphasis on the usual processing rather than on the abnormal processing. We have chosen the opposite solution because we want the definitions of the exceptions X_i and the variables V_i^j to precede their uses in B_0. We hereby follow the example of LOTOS' "**let**" and "**hide**" operators. □

Remark
At present, we see no practical reasons for allowing exception overloading. We can therefore require that all exceptions X_i declared in a "**trap**" operator are pairwise distinct. □

We must also extend process definitions with formal exception parameters, enclosed in braces:

$$\begin{array}{l} \textbf{process}\ P\ [...]\ (...)\ \{X_1\ [V_1^1 : T_1^1, ..., V_{m_1}^1 : T_{m_1}^1], ..., X_n\ [V_1^n : T_1^n, ..., V_{m_n}^n : T_{m_n}^n]\} \\ \quad B \\ \textbf{endproc} \end{array}$$

Remark
This is similar to ADA's "**raises** $X_1, ..., X_n$" clause which gives the list of exceptions raised by a given function or procedure. □

In this construct, the exceptions X_i are only visible in the process body B and the variables V_i^j are only visible in the "**raise** X_i" operators occurring in B.

To ensure symmetry with gate and value parameters, we also must extend process instantiations in order to allow actual exception parameters:

$$B \ ::= \ ...$$

$$| \quad P\ [...]\ (...)\ \{X_1, ..., X_n\}$$

The abbreviations proposed in Sections 3.18 and 3.19 should also apply to exception parameters. In particular, the clause "$\{X_1, ..., X_n\}$" can be omitted if there are no exception parameters or if the actual exception parameters have the same name as the formal ones.

As regards the dynamic semantics, we extend the transition relation of LOTOS. Besides transitions of the form "$B_1 \xrightarrow{G\ v_1,...,v_n} B_2$", we allow transitions of the form "$B_1 \xrightarrow{\diamond\ X\ v_1,...,v_n} B_2$", 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 "$\diamond\ X\ v_1, ..., v_n$". In the latter case, B_2 should be always equal (or, at least, strongly equivalent) to "**stop**".

The existing definition of label equality in LOTOS is slightly extended: two labels are equal if they carry the same gate identifier and the same offers, or the same exception identifier and the same offers. Therefore, if two labels are equal, either both of them have the "$\diamond$" symbol, or none of them has this symbol.

The dynamic semantics of the "**raise**" and "**trap**" operators is defined by the following rules:

1. raise of an exception:

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

2. normal execution:

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

3. catch of an exception:

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

4. propagation of an uncaught exception:

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

Remark

We make the following comments on this semantics:

- The "**raise**" operator is the only way to generate a "$\diamond$"-transition; the standard action-prefix operator of LOTOS does not generate "$\diamond$"-transition.
- When an exception is caught, the control transfer is atomic, thus not visible from the outside.
- In non-deterministic choices, the proposed semantics gives no priority to exceptions versus "ordinary" actions. Especially, the following behaviour expression:

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

  cannot be reduced to:

$$\textbf{raise}\ X$$

- We can justify why it is necessary to distinguish between exceptions and gates, as well as between "$\diamond$-transitions and ordinary transitions. It would not be possible to unify "**raise** X" and "X ; **stop**". For instance, let's consider the following behaviour expression:

```
trap
   X' → B'
in
   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).

- Because the exceptions X_i declared in a "**trap**" operator are only visible in B_0 and because they are necessarily caught by the "**trap**" operator, an exception can not escape out of its scope (a problem that exists in ML). Our proposal ensures that exceptions never propagate outside of the scope of their declaration (this is checked statically).
- If a LOTOS process is recursive through the first argument of a "**trap**" operator, this means that the process is re-instantiated every time the exception is raised.

- If a LOTOS process is recursive through the second argument of a "**trap**" operator, this means that, for each instance of the recursive process, an exception handler will be created and "stacked". This is called *dynamic scoping* and is similar to the "**setjmp/longjmp**" mechanism of the C language. On the other hand, ADA relies on *static scoping*: all the instances of a recursive process share the same exception handler.

□

Remark
In the "**trap**" operator, it might be desirable to add an "**otherwise** $\rightarrow B_{n+1}$" clause before the "**in**" keyword. This clause (similar to ADA's "**when others**" clause) would catch all exceptions raised by the normal behaviour B_0, but declared on the outside of the "**trap**" statement. Since there can be several such exceptions, their values could not be referenced in B_{n+1}. □

Because the above semantics keeps the rules for parallel composition unchanged, and because the notion of label equality has been extended to include exceptions, we can infer that the above semantics only allows *synchronous termination*: in a parallel composition "B_1 |[...]| B_2", if B_1 raises an exception, it will not occur unless B_2 also raises the same exception, with the same values. In some cases, this is not realistic: for instance, if B_1 signals a lack of memory or a "no_match" error in a "**case**" statement, we should not expect B_2 to raise the same exception at the same time.

As suggested in [QA92] and [Que96a, Annex F], it is also desirable to provide for *asynchronous termination*: the execution of "B_1 |[...]| B_2" is aborted as soon as either B_1 or B_2 raises an exception.

We believe that it is desirable to allow both synchronous and asynchronous termination, as synchronous termination already exists in LOTOS (the "**exit**" operator) and it can be useful for compositionality reasons (especially for constraint-oriented style).

For instance, the proposal in [Que96a, Annex F] allows synchronous termination (with the "**exit**" operator) and asynchronous termination (with user-defined exception names).

We slightly extend this proposal by allowing both synchronous and asynchronous termination on user-defined exception names. For doing so, we extend LOTOS parallel composition operator to include, besides the list of gates $G_1, ..., G_m$ to be synchronized, a list of exceptions $X_1, ..., X_n$ to be synchronized. The proposed syntax is:

$$B_1 \;\; |[G_1, ..., G_m \;\;|\;\; X_1, ..., X_n]| \;\; B_2$$

In this schema, the termination gate δ of LOTOS becomes an exception identifier: this is perfectly elegant, since the intended meaning of δ is rather an exception than a gate. As in LOTOS, the "δ" gate is never mentioned in the exception list $X_1, ..., X_n$ but always synchronized by the parallel operator. The dynamic semantics of the extended parallel operator is defined by the following rules:

1. non-synchronized gate

$$\frac{(B_1 \xrightarrow{L} B_1') \;\wedge\; (L \text{ has the form } G...) \;\wedge\; (G \notin \{G_1, ..., G_m\})}{(B_1 \;\; |[G_1, ..., G_m \;\;|\;\; X_1, ..., X_n]| \;\; B_2) \xrightarrow{L} (B_1' \;\; |[G_1, ..., G_m \;\;|\;\; X_1, ..., X_n]| \;\; B_2)}$$

   (add a similar rule for B_2).

2. synchronized gate

$$\frac{(B_1 \xrightarrow{L} B_1') \;\wedge\; (B_2 \xrightarrow{L} B_2') \;\wedge\; (L \text{ has the form } G...) \;\wedge\; (G \in \{G_1, ..., G_m\})}{(B_1 \;\; |[G_1, ..., G_m \;\;|\;\; X_1, ..., X_n]| \;\; B_2) \xrightarrow{L} (B_1' \;\; |[G_1, ..., G_m \;\;|\;\; X_1, ..., X_n]| \;\; B_2')}$$

3. non-synchronized exception

$$\frac{(B_1 \xrightarrow{L} B_1') \ \wedge \ (L \ has\ the\ form\ X...) \ \wedge \ (X \notin \{\delta, X_1, ..., X_n\})}{(B_1' \ \ |[G_1, ..., G_m \ \ | \ \ X_1, ..., X_n]| \ \ B_2) \xrightarrow{L} \textbf{stop}}$$

(add a similar rule for B_2).

4. synchronized exception

$$\frac{(B_1 \xrightarrow{L} B_1') \ \wedge \ (B_2 \xrightarrow{L} B_2') \ \wedge \ (L \ has\ the\ form\ X...) \ \wedge \ (X \in \{\delta, X_1, ..., X_n\})}{(B_1 \ \ |[G_1, ..., G_m \ \ | \ \ X_1, ..., X_n]| \ \ B_2) \xrightarrow{L} \textbf{stop}}$$

Remark
One might wonder whether the "**hide**" operator should also be extended with exception names. We believe that this is not necessary: when an exception X is caught by a "**trap**" operator, no X-transition can be observed from the outside; therefore, no third-party can interfere with this exception. This remark will turn to be essential in the sequel, in particular when modelling the enabling and disabling operators of LOTOS using exceptions. □

Remark
It might be suitable to introduce a renaming operator for exceptions, similar to the "**rename**" operator for gates (see Section 3.7). □

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:

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

   Remark
   If an "**exit**" operator has an "**any**" clause, it should be expanded into an "**any**" clause of the "**raise**" operator. □

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

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

   Remark
   In LOTOS, the ">>" operator is a primitive one, since it cannot be exactly expressed using parallel composition and hiding. The reason for this is the problem to give different names to the δ gate in case of nested ">>" operators (see [Gar89, chapter 2] for a discussion). Introducing the "**trap**" operator solves this problem elegantly: it is not necessary to hide "δ", because no δ-event can be observed from the outside if the exception is caught. □

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

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

where $\mathcal{F}$ denotes a list of "**any** T" clauses compatible, in number and types, with the functionality of B_1.

Remark
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.
- Omitting the "**exit** $\mathcal{F}$" 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.
- 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**").
- 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
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
Currently, functionality rules only deal with the "δ" exception. They should be extended to deal with all exceptions. □

3.14 Unifying the ";" and ">>" operators

LOTOS has two sequential composition operators: the action prefix operator (";") and the enabling operator (">>"). We believe that these operators are not satisfactory, for the following reasons:

1. The distinction between ";" and ">>" is often felt troublesome by novice users, who naturally tend to write incorrect behaviour expressions of the form "B_1 ; B_2". This is a major reason why Lotos has a steep leaning curve: even for the most intuitive concept of sequential composition, Lotos is incompatible with the usual notations of sequential programming languages. Other languages (for instance Esterel, Acp, Theoretical Csp) avoid these problems by offering sequential composition operators that are compatible with the existing practice.

2. From an algebraic point of view, the two Lotos operators ";" and ">>" are weak:

   - The former is asymmetrical, because its left argument cannot be a behaviour expression.
   - The latter is symmetrical and associative in absence of "**accept**" clause; however, in presence of "**accept**" clauses, it is not associative, because of different variable scopes. For instance:

   $$B_1 \;\texttt{>>}\; \textbf{accept}\; V_1 : T_1 \;\textbf{in}\; (B_2 \;\texttt{>>}\; \textbf{accept}\; V_2 : T_2 \;\textbf{in}\; B_3)$$

   is not identical to:

   $$(B_1 \;\texttt{>>}\; \textbf{accept}\; V_1 : T_1 \;\textbf{in}\; B_2) \;\texttt{>>}\; \textbf{accept}\; V_2 : T_2 \;\textbf{in}\; B_3$$

   because, in the latter case, V_1 is not visible in B_3.

3. Moreover, the two Lotos operators ";" and ">>" are not symmetrical with respect to variable scope rules:

   - With the former, the names and values of the variables declared on the left-hand side of the ";" operator are implicitly passed to the right-hand side. For instance, in "G ?X : T ; B", variable X is visible in B.
   - With the latter, neither the names nor the values of the variables declared on the left-hand-side of the ">>" operator are passed to the right-hand side. For instance, in "G ?X : T >> B, variable X is not visible in B. However, values (but not names) can be passed to the right-hand side: this has to be done explicitly, using the arguments of the "**exit**" operator.

4. The ">>" operator involves a rendez-vous on the termination gate "δ", thus leading to an "**i**" action when the enabling operator is used. On the other hand, the action-prefix operator is atomic and does not generate "**i**" actions.

   From a theoretical point of view, it is unpleasant to notice that the ">>" operator has no neutral element.

   From a practical point of view, the generation of "**i**" actions has the bad effect of increasing the size of the labelled transition systems generated from Lotos programs. This generates (or contributes to generate) 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 warned not to use the ">>" operator "too much"!

5. The existing LOTOS operators are not convenient for specifying certain situations. Let's consider the following "distributed query" problem. Let's assume three distributed processes P_1, P_2, and P_3. Each process P_i contains an integer value V_i. In order to compute the sum $V_1 + V_2 + V_3$, it is necessary to ask each process P_i to send his value V_i. If the processes are asked sequentially, the solution is easy:

$$\begin{array}{l} G_1 \;\; ?V_1 : \mathbf{int} \;\; ; \\ G_2 \;\; ?V_2 : \mathbf{int} \;\; ; \\ G_3 \;\; ?V_3 : \mathbf{int} \;\; ; \\ \mathbf{exit} \; (V_1 + V_2 + V_3) \end{array}$$

However, if one does not want to fix the order in which the processes are asked, the solution is much more subtle and within the reach of only experienced LOTOS users:

$$\begin{array}{l} (\\ \quad (G_1 \;\; ?V_1 : \mathbf{int} \;\; ; \;\; \mathbf{exit} \; (V_1, \mathbf{any\ int}, \mathbf{any\ int})) \\ \quad ||| \\ \quad (G_2 \;\; ?V_2 : \mathbf{int} \;\; ; \;\; \mathbf{exit} \; (\mathbf{any\ int}, V_2, \mathbf{any\ int})) \\ \quad ||| \\ \quad (G_3 \;\; ?V_3 : \mathbf{int} \;\; ; \;\; \mathbf{exit} \; (\mathbf{any\ int}, \mathbf{any\ int}, V_3)) \\ ) \\ \gg \; \mathbf{accept} \; V_1, V_2, V_3 : \mathbf{int\ in\ exit} \; (V_1 + V_2 + V_3) \end{array}$$

This solution is not compositional: the arguments of the "**exit**" depend upon the number of parallel processes. Adding a new process requires to modify the definition of existing processes. Each process P_i has to be aware of the total number of processes and of its own index i within the list of processes. Should the variables V_i have different types, the situation should be even worse since each process should know about the functionality of other processes.

Although the ellipsis notation ("...") proposed for gate typing might reduce the complexity of this solution, it would be better to have a simpler syntax to describe this behaviour, e.g.:

$$\begin{array}{l} (\\ \quad G_1 \;\; ?V_1 : \mathbf{int} \\ \quad ||| \\ \quad G_2 \;\; ?V_2 : \mathbf{int} \\ \quad ||| \\ \quad G_3 \;\; ?V_3 : \mathbf{int} \\ ) \;\; ; \\ \mathbf{exit} \; (V_1 + V_2 + V_3) \end{array}$$

For these reasons, it would be desirable to have a *single* operator for sequential composition that satisfies the following requirements:

1. This operator should be *symmetrical*: its arguments should be two behaviour expressions.

2. The rules for making the variables declared on the left-hand argument visible in the right-hand argument should be both intuitive and compatible with the existing rules of LOTOS for action prefix and enabling.

3. It should be *associative*, even in presence of variable passing: to keep the language simple and user-friendly, variable scoping should not rely on the place of parentheses.

4. It should be *atomic*: sequential composition should not generate "**i**" transitions.

5. It should have "**exit**" as a neutral element.

6. It should be *expressive* enough to solve the distributed query problem elegantly.

Remark
In his thesis [Bri88], Ed Brinksma acknowledged the need for improving sequential composition in LOTOS. He proposed to merge both operators ";" and ">>" from a syntactical point of view. An outline of his proposal can be found in Annex B. Basically, his solution 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:

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

Actually, Brinksma's proposal is more sophisticated, in order to avoid potential ambiguities created by the fact that the ";" operator becomes overloaded (it has two possible profiles, e.g., "$G\ ;\ B_0$" and "$B_1\ ;\ B_2$"). For instance, there is an ambiguity between:

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

and:

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

Such problems are avoided by the bracketed syntax proposed by Ed Brinksma[4].

From our point of view, Brinksma's proposal is not satisfactory: because it remains at the level of syntactic unification, it does not satisfy the above requirements. There are still two distinct sequential composition operators (although they share a unified syntax). Moreover, as Brinksma keeps the existing semantics of the ">>" operator unchanged, his proposal still lacks associativity, atomicity and neutral element. 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. □

In Section 3.13, we have seen that the "**exit**" and ">>" operators of LOTOS (and also "[>") can be obtained as particular cases of the exception mechanism. We must examine whether we could not build, on top of the exception mechanism, a "better" sequential composition operator that matches the above requirements.

First, let's consider a simple case, in which action denotations do not have experiment offers. We modify the syntax of behaviour expressions in two ways: first, action denotations (without action denotations) become plain behaviour expressions; second, we introduce a symmetrical sequential composition operator:

$$\begin{array}{lcl} B & ::= & ... \\ & | & G \\ & | & B_1\ ;\ B_2 \end{array}$$

[4] A simple solution to this problem would be to prohibit boolean guards in action denotations when no offers are present. In such case, users should write "$G\ [V]\ ;\ B$" instead of "$[V] \rightarrow G\ ;\ B$".

The dynamic semantics of the action "G" is defined as follows:

$$\frac{true}{G \xrightarrow{G} (\textbf{raise}\ \delta)}$$

meaning that an action "G" involves two successive steps, the action itself followed by a continuation passing modelled by an exception on the δ gate: "$G \xrightarrow{G} \textbf{raise}\ \delta \xrightarrow{\diamond\delta} \textbf{stop}$".

We define the sequential composition operator as a shorthand for a "**trap**" operator:

$$B_1\ ;\ B_2 =_{def} \left(\begin{array}{l} \textbf{trap} \\ \quad \delta \rightarrow B_2 \\ \textbf{in} \\ \quad B_1 \\ \textbf{endtrap} \end{array} \right)$$

Its semantics can also be defined by a set of rules:

$$\frac{(B_1 \xrightarrow{L} B_1') \wedge (L\ has\ the\ form\ G...)}{(B_1\ ;\ B_2) \xrightarrow{L} (B_1'\ ;\ B_2)}$$

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

$$\frac{(B_1 \xrightarrow{L} B_1') \wedge (L\ has\ the\ form\ X...) \wedge (X \neq \delta)}{(B_1\ ;\ B_2) \xrightarrow{L} B_1'}$$

Our syntax is obviously a superset of the existing LOTOS syntax: the action prefix "$G\ ;\ B_0$" can be obtained as a special case of "$B\ ;\ B_0$" where $B = G$. We also preserve the semantics of LOTOS' action prefix operator: according to our proposal, a behaviour expression of the form "$G\ ;\ B_0$" will be parsed as "$(G)\ ;\ (B_0)$" and will have the same behaviour as in standard LOTOS.

Remark

The ";" operator has a neutral element: "**exit**", which is defined as "**raise** δ" (see Section 3.13). For all B, we have:

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

□

Remark

"**stop**" has interesting algebraic properties. On the left-hand side of ";", it is absorbing:

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

On the right-hand side, it can be used as a way to removes the trailing "δ" continuation that follows an action. In particular, "G" is different from "$G\ ;\ \textbf{stop}$" since the LTS of the former is "$G \xrightarrow{G} \textbf{raise}\ \delta \xrightarrow{\diamond\delta} \textbf{stop}$" whereas the LTS of the latter is "$G \xrightarrow{G} \textbf{stop}$". □

Remark
As explained in Section 3.13, the enabling operator of LOTOS can still be obtained as a shorthand notation of the "**trap**" operator. It can also be obtained as a shorthand of the ";", which proves that, even in absence of "**trap**", our sequential composition is more primitive:

$$B_1 \;\texttt{>>}\; B_2 =_{def} B_1 \;\;;\;\; \mathbf{i} \;\;;\;\; B_2$$

□

At present, let's consider the general case, in which action denotations do not have experiment offers. First to be solved is the problem of variable scoping. In order to maintain compatibility with standard LOTOS, it is necessary to determine how the variables declared on the left-hand side of a ";" operator (especially in "?"-offers) can be made visible on the right-hand side. For instance, in the following behaviour expression:

$$(G \;\; ?X : T) \;\; ; \;\; (G' \;\; ?X' : T') \;\; ; \;\; B$$

the variables X and X' should be visible in B.

In some other cases, we have a greater freedom to determine scope rules, since there no such constraints of compatibility with LOTOS. But there are still other criteria to be satisfied, such as associativity and abstraction.

We define a *functionality function* "$\mathcal{F}(B)$" which associates, to each behaviour expression B the list of variable bindings exported by B, i.e., the list of variables whose value is available when B terminates by raising a δ exception *whatever the place from which the δ exception is raised.*

Remark
In fact, the function $\mathcal{F}$ should perhaps be extended in two ways, in order:

- to integrate the possibility that a behaviour never terminates (i.e., "**noexit**" in standard LOTOS)
- to deal with all exceptions, not only the "δ" exception (this would provide static semantics checkings for all exceptions).

□

This function is defined by structural induction on the syntactic structure of behaviour expressions. For instance, we define:

- $\mathcal{F}(\textbf{stop}) = \emptyset$
- $\mathcal{F}(G \;\; ?V_1 : T_1, ..., V_n : T_n) = \{V_1 : T_1, ..., V_n : T_n\}$
- For non-deterministic choice (as well as for "**if**" and "**case**" operators, and all other forms of behaviour expressions which terminate in various places), we define:

  $$\mathcal{F}(B_1 \;\; [] \;\; B_2) =_{def} (\mathcal{F}(B_1) \cap \mathcal{F}(B_2))$$

  meaning that only those variables visible in both operands are visible in the non-deterministic composition. For instance:

  $$\mathcal{F}(G \;\; ?V_1 : T_1 \;\; ?V_2 : T_2 \;\; [] \;\; G \;\; ?V_1 : T_1) = \{V_1 : T_1\}$$

  If there are several variables with the same names but different types, they are excluded from the intersection. For instance:

  $$\mathcal{F}(G \;\; ?V : T_1 \;\; [] \;\; G \;\; ?V : T_2) = \emptyset$$

- For parallel composition, we define:

$$\mathcal{F}(B_1 \ \ ||| \ \ B_2) =_{def} (\mathcal{F}(B_1) \cup \mathcal{F}(B_2))$$

meaning that the variables computed by each operand are all available when the parallel execution terminates. This solves the aforementioned "distributed query" problem. In the following example:

$$((G \ \ ?X : T) \ \ ||| \ \ (G' \ \ ?X' : T')) \ \ ; \ \ B$$

variables X and X' are visible in B.

If there are several variables with the same names but different types, they are excluded from the union. For instance:

$$\mathcal{F}(G \ \ ?V : T_1 \ \ ||| \ \ G \ \ ?V : T_2) = \emptyset$$

- For the instantiation of a process P that terminates properly (using a "δ" exception), the variables defined in the body of P are local and should not be exported. For instance, in the following expression:

$$P \, [...] \, (...) \ \ ; \ \ B$$

the local variables of P should not be visible in B.

With respect to the other operators, we see two main approaches for defining the function "$\mathcal{F}$":

Implicit continuation passing: in this approach, all variables declared on the left-hand side of an "**;**" operator are automatically visible on the right-hand side. For sequential composition, we define:

$$\mathcal{F}(B_1 \ \ ; \ \ B_2) =_{def} (\mathcal{F}(B_1) - \mathcal{F}(B_2)) \cup \mathcal{F}(B_2)$$

meaning that the variables visible in "$B_1 \ \ ; \ \ B_2$" are those of B_2 and those of B_1 which are not redefined in B_2.

For the "**let**" operator, we can adopt the following definition:

$$\mathcal{F}(\textbf{let } V_1 : T_1 \ \ \texttt{:=} \ \ E_1, ..., V_n : T_n \ \ \texttt{:=} \ \ E_n \textbf{ in } B) =_{def} \mathcal{F}(B)$$

if we want to ensure that the variables V_i are only visible in B. However, it would be difficult to explain why the variables declared in "**?**"-offers pass across the "**;**" operator, whereas the variables declared "**let**" statements don't. Therefore we could introduce a new "functional assignment" operator noted:

$$V : T \ \ \texttt{:=} \ \ E$$

which is equivalent to "**raise** δ" and such that:

$$\mathcal{F}(V : T \ \ \texttt{:=} \ \ E) = \{V : T\}$$

This operator would generalize the existing "**let**" operator:

$$\textbf{let } V_1 : T_1 \ \ \texttt{:=} \ \ E_1, ..., V_n : T_n \ \ \texttt{:=} \ \ E_n \textbf{ in } B =_{def} (V_1 : T_1 \ \ \texttt{:=} \ \ E_1) \ \ ; \ \ ... \ \ ; \ \ (V_n : T_n \ \ \texttt{:=} \ \ E_n) \ \ ; \ \ B$$

and would be helpful to factorize computations which would otherwise turn to combinatorial explosion, e.g.:

```
if E then
  X : T  :=  E1
else
  G ?X : T
endif
;
if E' then
  X' : T  :=  E'1
else
  G ?X' : T
endif
;
if E'' then
  X'' : T  :=  E''1
else
  G ?X'' : T
endif
;
G !X  !X'  !X''
```

A key feature of this approach is that there is no other mean to "hide" local variables that encapsulating in a process definition.

Explicit continuation passing: in this approach, by default, all variables declared on the left-hand side of an ";" operator are not visible on the right-hand side. An additional mechanism must be added to specify which variables pass across the ";" operator.

A first solution consists in using Brinksma's "**init**" clause (which is similar to LOTOS' "**accept**" clause). For instance, if one writes:

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

only the variables $V_1, ..., V_n$ are passed to B_2; the other variables of B_1 are not transmitted. Although this scheme provides abstraction for all local variables of B_1, there is one main issue. There is no way under this scheme to allow an expression such as "$(G \ ?V : T) \ ; \ B$", in which V is visible in B. An action prefix operator is also needed: this is why Brinksma's proposal [?] has two sequential composition operators.

A second solution is to specify which variables are exported in the "**exit**" statement itself, thus avoiding the "**init**" clause. The "**exit**" statement should be responsible for declaring the variables whose names and values will be passed in the continuation. We extend the syntax of "**exit**" as follows:

$$\begin{array}{lcl} B & ::= & ... \\ & | & \textbf{exit} \ (V_1 : T_1, ..., V_n : T_n) \end{array}$$

and we define:

$$\mathcal{F}(\textbf{exit } (V_1 : T_1, ..., V_n : T_n)) =_{def} \{V_1 : T_1, ..., V_n : T_n\}$$

For instance, in the following behaviour expression:

$$(G \ \texttt{?}X : T \ \ \texttt{;} \ \ \textbf{exit } (X' : T \ \ \texttt{:=} \ \ E)) \ \texttt{>>} \ B$$

the variable X' (but not X) should be visible in B.

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 $P_1, ..., P_n$. □

3.15 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 3.13.

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:

$$\begin{array}{l} P[...](E_1, ..., E_n) \\ \textbf{where} \\ \textbf{process } P[...](V_1 : T_1, ..., V_n : T_n)... \\ \quad \textbf{trap} \\ \quad\quad \theta \ V_1' : T_1, ..., V_n' : T_n \rightarrow P[...](V_1', ..., V_n') \\ \quad \textbf{in} \\ \quad\quad B_0 \\ \quad \textbf{endtrap} \\ \textbf{endproc} \end{array}$$

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 3.13 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. □

3.16 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 [BL95]:

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.

3.17 Simplifying process definitions

In this section, 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).

In the data part, it has been agreed that functions of the user data language should have "**in**" and "**out**" parameters, mixed in any order. The rationale for this design choice can be found in [GS95] (this is especially suitable to interface foreign languages such as Idl and other programming languages).

To maintain a clear symmetry between the behaviour and the user data language [SG95] of E-Lotos, it would be also suitable that processes have "**in**" and "**out**" parameters, mixed in any order. We suggest to introduce "**out**" parameters in process definitions to replace functionality declarations. We propose to replace process definitions such as:

process $P[G_1, ..., G_n](V_1 : T_1, ..., V_m : T_m)$: **exit** $(T'_0, ..., T'_p)$:=
  B
endproc

with:

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

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:

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

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:

  $$[V] \rightarrow \textbf{stop}$$
  $$[]$$
  $$[\neg V] \rightarrow \textbf{exit}$$

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

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

  and:

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

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):

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

Remark
Our proposal should be extended to integrate exceptions, as described in Section 3.13. Should exceptions be introduced, the the "**out**" parameters of a process definition would be nothing but the parameters of the δ exception raised at the end of the process. □

3.18 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 actual gate parameter lists.

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

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[5] "$G'_i = G_i$".

   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 :=
```

[5] this is an exclusive "or"

```
      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]. □

3.19 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. 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 3.18 and 3.19 are demonstrated in Annexes C and D.

4 Definition of the user language

4.1 Schema overview

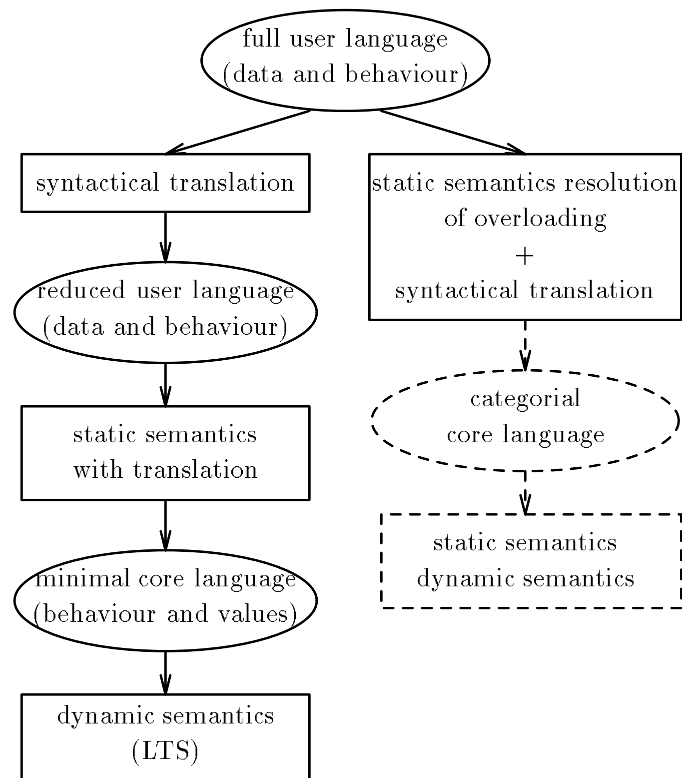


Figure 1: The schema of translations

4.2 Notations

We present the notations used in the presentation of abstract syntaxes.

The classes of *terminals identifiers* of the language are:

identifier domain	*meaning*	*abbreviations*
SCon	special constants	K
Var	variable identifiers	V
Typ	type identifiers	T
ChName	channels and/or exception identifiers	Γ, Ξ
Con	constructor identifiers	C
Fun	function identifiers	F
Proc	process identifiers	Π
Gat	gate and/or exception identifiers	G, X

The special constants are values of built-in types (**bool**, **int**) provided automatically to user.

Remark: The unification of exceptions and gates is suitable. However, to avoid some confusions, and to preserve the user sense about exception, we give them to different notation for their identifiers in syntax: X for exceptions and G for gates. We mention that this identifiers belong over the same domain. The same remark is true for exceptions and gate type identifiers.

We will also define the following *classes of non-terminal* symbols:

symbol domain	*meaning*	*abbreviations*
Decl	declaration	D
Pat	patterns	P
Match	match expression	M
Exp	extended expression	E
Behav	behavior expression	B
Offr	offer	O

We use the following notations for set of indexes:

notation	*meaning*
Σ	non-empty set of indexes (its meaning is context dependent)
Σ^p	non-empty set of indexes with p elements
$\Sigma_{0..n}$	non-empty subset of $\{0..n\}$
$\Sigma^p_{0..n}$	non-empty subset of $\{0..n\}$ having p elements

To express the abstract syntax of user language we used the following meta-symbols are use to describe the syntax:

symbol	meaning
::=	defined to be
|	alternatively
[ ]	the enclosed syntactic unit is optional—may occur zero or one time

4.3 Abstract syntax of the full user language

4.3.1 Declarations

D ::= **type** T **is** T' **endtype** — *type synonym* (D_f1)

| **type** T **is** — *type declaration* (D_f2)
$C_1 [(V_1^1 : T_1^1, ..., V_{n_1}^1 : T_{n_1}^1)]$
...
$C_p [(V_1^p : T_1^p, ..., V_{n_p}^p : T_{n_p}^p)]$
endtype $[T]$

| **exception** Ξ **is** — *exception declaration* (D_f3)
$[(V_1 : T_1, ..., V_n : T_n)]$
endexc

| **channel** Γ **is** — *channel declaration* (D_f4)
$(V_1^1 : T_1^1, ..., V_{n_1}^1 : T_{n_1}^1)$ — *profile list of gate typing*
...
$(V_1^p : T_1^p, ..., V_{n_p}^p : T_{n_p}^p)$
endchan

| **function** $F [X_1 : \Xi_1, ..., X_p : \Xi_p]$ — *function declaration* (D_f5)
([**in** | **out**] $V_1 : T_1, ...,$ [**in** | **out**] $V_n : T_n$)
$[: T]$ **is**
E
endfunc $[F]$

| **process** $\Pi [G_1 : \Gamma_1, \ldots, G_p : \Gamma_p]$ — *process declaration* (D_f6)
([**in** | **out**] $V_1 : T_1, ...,$ [**in** | **out**] $V_n : T_n$)
[**:noexit** | **:exit** $(T'_1, ..., T'_m)$] **is**
B
endproc $[\Pi]$

| $D_1 D_2$ — *sequence of declaration* (D_f7)

| — *empty declaration* (D_f8)

Remarks:

1. In $(D2)$, $p \geq 0$; if $p = 0$ then T could be considered as an external type.
2. In $(D2)$ and $(D5)$, as in LOTOS, constructors and functions can be declared to be infixed.
3. In $(D5)$, the kind of formal parameter (i.e. "**in**") is optional.

4. In $(D5-7)$, the expressions E (respectively the behaviour B) may contain F (respectively Π) calls.

5. $(D7-8)$ are useful only for module system.

4.3.2 Patterns

$P ::= K$ *constant* (P_f1)

$\mid V$ *variable* (P_f2)

$\mid$ **any** T (P_f3)

$\star \mid C([V_1 \texttt{:=}]P_1, ..., [V_n \texttt{:=}]P_n[, \ldots])$ *constructed pattern* (P_f4)

$\mid P_0$ **of** T *typed pattern* (P_f5)

Remarks:

1. In $(P4)$, if formal parameters are specified, parameters not explicitely specified are denoted by "$\ldots$".

4.3.3 Match expressions

$M ::= E \texttt{::} P$ (M_f1)

$\mid M_1$ **when** E (M_f2)

$\mid M_1$ **and** M_2 (M_f3)

$\mid M_1$ **or** M_2 (M_f4)

4.3.4 Value expressions

$E ::= K$ *constant denotation* (E_f1)

$\mid V$ *value variable* (E_f2)

$\star \mid C([V_1 \texttt{:=}]E_1, ..., [V_n \texttt{:=}]E_n[, \ldots])$ *constructor application* (E_f3)

$\star \mid F \texttt{[}[X_1 \texttt{:=}]X'_1, \ldots, [X_p \texttt{:=}]X'_p[, \ldots]\texttt{]}$ *function call* (E_f4)
$([V_1 \texttt{:=}]E_1 \mid [V_1 \texttt{=:\&}]P_1, ..., [V_n \texttt{:=}]E_n \mid [V_n \texttt{=:\&}]P_n[, \ldots])$

| **case** *general case expression* $(E_f 5)$

$M_0 \rightarrow E_0$

...

$M_p \rightarrow E_p$

⋆ [**otherwise** E_{p+1}]

endcase

⋆ | **case** E' **in** *usual case expression* $(E_f 6)$

$P^0_1, ..., P^0_{n_0}$[**when** E_0] $\rightarrow E'_0$

...

$P^p_1, ..., P^p_{n_p}$[**when** E_p] $\rightarrow E'_p$

[**otherwise** E'_{p+1}]

endcase

⋆ | E **match** P *match expression* $(E_f 7)$

⋆ | **if** E_0 **then** E'_0 *conditional expression* $(E_f 8)$

elsif E_1 **then** E'_1

...

elsif E_n **then** E'_n

[**else** E'_{n+1}]

endif

⋆ | E_0 **andthen** E_1 *logical expression* $(E_f 9)$

⋆ | E_0 **orelse** E_1 *logical expression* $(E_f 10)$

| **raise** X[([V_1:=]E_1, ..., [V_n:=]E_n)] *exception raise* $(E_f 11)$

| **trap** *trap exceptions* $(E_f 12)$

$X_1[(V^1_1 : T^1_1, ..., V^1_{p_1} : T^1_{p_1})] \rightarrow E_1$

...

$X_n[(V^n_1 : T^n_1, ..., V^n_{p_n} : T^n_{p_n})] \rightarrow E_n$

in E

endtrap

| **local** $V_1 : T_1, \ldots, V_n : T_n$ **in** E **endloc** *local declaration* $(E_f 13)$

⋆ | **let** $V_1 : T_1$:=$E_1, \ldots, V_n : T_n$:=E_n **in** E [**endlet**] *let expression* $(E_f 14)$

| P := E *variable assignment* $(E_f 15)$

	$\mid$	E_1 ; E_2	*expression continuation* (E_f16)
	$\mid$	E_0 = E_1	*equality expression* (E_f17)
	$\star\mid$	E_0 <> E_1	*non-equality expression* (E_f18)
	$\mid$	E_0 . V_0	*select expression* (E_f19)
	$\mid$	E_0 . { V_1 := E_1, ..., V_n := E_n }	*update expression* (E_f20)
	$\mid$	**loop** [V_1 : T_1 := $E_1, \ldots, V_n$: T_n := E_n] **in** E_0 **endloop**	*loop expression* (E_f21)
	$\mid$	**continue** [V_1 := $E_1, \ldots, V_n$:= E_n]	(E_f22)
	$\mid$	E_0 **of** T	*layered value* (E_f23)

Remarks:

1. In ($E3-6$), constructors and functions without arguments, or constructors and functions infixed, are included by these rules.
2. In ($E3-4$), actual parameters corresponding to "**in**" formal parameters must be expressions values (without variable binding). Actual parameters corresponding to "**out**" formal parameters must be patterns.

   Also, the static semantics will reject calls merging labeled and unlabeled actual parameters.

4.3.5 Behaviour expressions

B	::=	G ([V_1 :=] E_1 | [V_1 =:&] $P_1, \ldots,$ [V_n :=] E_n | [V_n =:&] P_n [, ...]) [[E_{bool}]]	*action* (B_f1)
	$\star\mid$	**raise** G ([V_1 :=] $E_1, \ldots,$ [V_n :=] E_n [, ...])	*exception raise* (B_f2)
	$\star\mid$	**stop**	*inaction* (B_f3)
	$\star\mid$	**exit** [(E_1 | **any** $T_1, \ldots, E_n$ | **any** T_n)]	*successful termination* (B_f4)
	$\mid$	P := E	*assignment* (B_f5)

	$\mid$	**trap**
		  $G_1[(V_1^1 : T_1^1, ..., V_{p_1}^1 : T_{p_1}^1)] \rightarrow B_1$
		  ...
			trap gates (B_f6)

$G_n[(V_1^n:T_1^n, ..., V_{p_n}^n:T_{p_n}^n)] \rightarrow B_n$
in B
endtrap

⋆ | **seq** B_0 ; ... ; B_n **endseq** — *sequential composition* (B$_f$7)

| B_1 ; B_2 — *sequential composition* (B$_f$8)

⋆ | G !E_1 | ?V_1:T_1 ... !E_1 | ?V_n:T_n [[E_{bool}]] ; B — *old action prefix* (B$_f$9)

⋆ | B_1 >> **accept** V_1:T_1 ... V_n:T_n **in** B_2 — *enabling* (B$_f$10)

⋆ | **alt** B_0 [] ... [] B_n **endalt** — *multiple non-deterministic choice* (B$_f$11)

⋆ | B_1 [] B_2 — *old choice operator* (B$_f$12)

⋆ | **choice** V_1:T_1, ..., V_n:T_n [] B_0 [**endch**] — *choice over values* (B$_f$13)

| P := **any** T — *choice value* (B$_f$14)

⋆ | **dis** B_1 [> B_2 **enddis** — *disabling* (B$_f$15)

| B_1 [> B_2 (B$_f$16)

| **par** G_1:Σ_1, ..., G_p:Σ_p — *general parallel composition* (B$_f$17)
B_0 || ... || B_n
endpar

⋆ | **par** — *parallel composition* (B$_f$18)
[G_1^0, ..., $G_{p_0}^0$] **for** B_0
|| ...
|| [G_1^m, ..., $G_{p_m}^m$] **for** B_m
endpar

| **par** V_1:T_1 ... V_n:T_n || B_0 [**endpar**] — *parallel over values* (B$_f$19)

⋆ | **par sync** [$G_0\#n_0$, ..., $G_p\#n_p$ | **all** | **none**] — *n among m synchronization* (B$_f$20)
B_0 || ... || B_m
endpar

⋆ | B_1 |[G_1, ..., G_p]| B_2 (B$_f$21)

$\star$ | B_1 || B_2 (B$_f$22)

$\star$ | B_1 ||| B_2 (B$_f$23)

| **case** *general case* (B$_f$24)
$M_0 \rightarrow B_0$
...
$M_p \rightarrow B_p$
$\star$ [**otherwise** B_{p+1}]
endcase

$\star$ | **case** E' **in** *usual case* (B$_f$25)
$P_1^0, ..., P_{n_0}^0$[**when** E_0] $\rightarrow B'_0$
...
$P_1^p, ..., P_{n_p}^p$[**when** E_p] $\rightarrow B'_p$
[**otherwise** B_{p+1}]
endcase

$\star$ | **if** E_0 **then** B_0 *conditional expression* (B$_f$26)
elsif E_1 **then** B'_1
...
elsif E_p **then** B'_p
[**else** B_{p+1}]
endif

$\star$ | $[E_{\text{bool}}]$ -> B_0 *guarded behavior* (B$_f$27)

| **local** $V_1 : T_1, \ldots, V_n : T_n$ **in** B **endloc** *local declaration* (B$_f$28)

$\star$ | **let** $V_1 : T_1$:= $E_1, \ldots, V_n : T_n$:= E_n **in** B [**endlet**] *variable binding* (B$_f$29)

| **loop** $[V_1 : T_1$:= $E_1, \ldots, V_n : T_n$:= $E_n]$ **in** *loop* (B$_f$30)
B_0
endloop

| **continue** $[V_1$:= $E_1, \ldots, V_n$:= $E_n]$ *loop continue* (B$_f$31)

| **hide** $G_1 : \Gamma_1, \ldots, G_n : \Gamma_n$ **in** B_0 [**endhide**] (B$_f$32)

| Π [[G_1 :=]$G'_1, \ldots,$ [G_p :=]G'_p[, ...]] *process instantiation* (B$_f$33)
([V_1 :=]E_1 | [V_1=:&]$P_1, \ldots,$ [V_n :=]E_n | [V_n=:&]P_n[, ...])

| **rename** *post-renaming* (B.34)

$G_0[(V_1^0 : T_1^0, \ldots V_{n_0}^0 : T_{n_0}^0)]$ **:=** $G'_0[([V_1'^0 \textbf{:=}] E_1'^0, \ldots, [V_{m_0}'^0 \textbf{:=}] E_{m_0}'^0)]$

...

$G_p[(V_1^p : T_1^p, \ldots V_{n_p}^p : T_{n_p}^p)]$ **:=** $G'_p[([V_1'^p \textbf{:=}] E_1'^p, \ldots, [V_{m_p}'^p \textbf{:=}] E_{m_p}'^p)]$

in B

endren

4.4 Abstract syntax of the reduced user language

The declarations, patterns, match expressions, and offers are the same of the full language. From the grammars for expressions 4.3.4 and behaviour expressions 4.3.5 of the full language all rules marked with $\star\,|$ are eliminated. The constructions corresponding to this rules will be "syntactically" translated in rules of reduced language in section 4.6.

4.4.1 Declarations

In the reduced language, all parameters of functions and processes must be decorated. This is made automatically by the "syntactical" translation function.

D ::= **type** T **is** T' **endtype** — *type synonym* (D_r1)

| **type** T **is**
  $C_1[(V_1^1 : T_1^1, ..., V_{n_1}^1 : T_{n_1}^1)]$
  ...
  $C_p[(V_1^p : T_1^p, ..., V_{n_p}^p : T_{n_p}^p)]$
endtype $[T]$ — *type declaration* (D_r2)

| **exception** Ξ **is**
  $[(V_1 : T_1, ..., V_n : T_n)]$
endexc — *exception declaration* (D_r3)

| **channel** Γ **is** — *channel declaration* (D_r4)
  $(V_1^1 : T_1^1, ..., V_{n_1}^1 : T_{n_1}^1)$ — *profile list of gate typing*
  ...
  $(V_1^p : T_1^p, ..., V_{n_p}^p : T_{n_p}^p)$
endchan

| **function** $F[X_1 : \Xi_1, ..., X_p : \Xi_p]$
    (**in** | **out** $V_1 : T_1$, ..., **in** | **out** $V_n : T_n$)
  $[: T]$ **is**
  E
endfunc $[F]$ — *procedure declaration* (D_r5)

| **process** $\Pi[G_1 : \Gamma_1, \ldots, G_p : \Gamma_p]$
    (**in** | **out** $V_1 : T_1$, ..., **in** | **out** $V_n : T_n$)
  [**:noexit** | **:exit** $(T'_1, ..., T'_m)$] **is**
  B
endproc $[\Pi]$ — *process declaration* (D_r6)

| $D_1 D_2$ — *sequence of declaration* (D_r7)

		empty declaration	$(\mathrm{D}_r 8)$

4.4.2 Patterns

P	$::=$	K	*constant*	$(\mathrm{P}_r 1)$
	$\mid$	V	*variable*	$(\mathrm{P}_r 2)$
	$\mid$	**any** T		$(\mathrm{P}_r 3)$
	$\mid$	$C(P_1, ..., P_n)$	*constructed pattern*	$(\mathrm{P}_r 4)$
	$\mid$	P_0 **of** T	*typed pattern*	$(\mathrm{P}_r 5)$

4.4.3 Match expressions

M	$::=$	$E::P$	$(\mathrm{M}_r 1)$
	$\mid$	M_1 **when** E	$(\mathrm{M}_r 2)$
	$\mid$	M_1 **and** M_2	$(\mathrm{M}_r 3)$
	$\mid$	M_1 **or** M_2	$(\mathrm{M}_r 4)$

4.4.4 Value expressions

E	$::=$	K	*constant denotation*	$(\mathrm{E}_r 1)$
	$\mid$	V	*value variable*	$(\mathrm{E}_r 2)$
	$\mid$	$C(E_1, ..., E_n)$	*constructor application*	$(\mathrm{E}_r 3)$
	$\mid$	$F[X'_1, \ldots, X'_p](E_1 \mid \&P_1, ..., E_n \mid \&P_n)$	*function call*	$(\mathrm{E}_r 4)$
	$\mid$	**case** $M_0 \rightarrow E_0$ ...	*general case expression*	$(\mathrm{E}_r 5)$

$M_p \rightarrow E_p$
endcase

| **raise** $X[(E_1, ..., E_n)]$ *exception raise* $(\mathrm{E}_r 6)$

| **trap** *trap exception* $(\mathrm{E}_r 7)$
$X_1[(V_1^1 : T_1^1, ..., V_{p_1}^1 : T_{p_1}^1)] \rightarrow E_1$
...
$X_n[(V_1^n : T_1^n, ..., V_{p_n}^n : T_{p_n}^n)] \rightarrow E_n$
in E
endtrap

| **local** $V_1 : T_1, \ldots, V_n : T_n$ **in** E **endloc** *local declaration* $(\mathrm{E}_r 8)$

| P := E *variable assignment* $(\mathrm{E}_r 9)$

| E_1 ; E_2 *expression continuation* $(\mathrm{E}_r 10)$

| E_0 = E_1 *equality expression* $(\mathrm{E}_r 11)$

| $E_0 . V_0$ *select expression* $(\mathrm{E}_r 12)$

| $E_0 . \{V_1 := E_1, ..., V_n := E_n\}$ *update expression* $(\mathrm{E}_r 13)$

| **loop** $[V_1 : T_1 := E_1, \ldots, V_n : T_n := E_n]$ **in** *loop expression* $(\mathrm{E}_r 14)$
E_0
endloop

| **continue** $[V_1 := E_1, \ldots, V_n := E_n]$ $(\mathrm{E}_r 15)$

| E_0 **of** T *layered value* $(\mathrm{E}_r 16)$

4.4.5 Behaviour expressions

B ::= $G(E_1 \mid \&P_1, \ldots, E_n \mid \&P_n)$ $[[E_{\mathrm{bool}}]]$ *action* $(\mathrm{B}_r 1)$

| P := E *assignment* $(\mathrm{B}_r 2)$

| **trap** *trap gates* $(\mathrm{B}_r 3)$
$G_1[(V_1^1 : T_1^1, ..., V_{p_1}^1 : T_{p_1}^1)] \rightarrow B_1$
...

$G_n[(V_1^n : T_1^n, ..., V_{p_n}^n : T_{p_n}^n)] \rightarrow B_n$
in B
endtrap

| B_1 ; B_2 — *sequential composition* (B_r4)

| P := **any** T — *choice value* (B_r5)

| B_1 [> B_2 — *disabling* (B_r6)

| **par** $G_1 : \Sigma_1, \ldots, G_p : \Sigma_p$ — *general parallel composition* (B_r7)
B_0 || ... || B_n
endpar

| **par** $V_1 : T_1, \ldots, V_n : T_n$ || B_0 [**endpar**] — *parallel over values* (B_r8)

| **case** — *general case* (B_r9)
$M_0 \rightarrow B_0$
...
$M_p \rightarrow B_p$
endcase

| **local** $V_1 : T_1, \ldots, V_n : T_n$ **in** B **endloc** — *local declaration* (B_r10)

| **loop** $[V_1 : T_1 := E_1, \ldots, V_n : T_n := E_n]$ **in** — *loop* (B_r11)
B_0
endloop

| **continue** $[V_1 := E_1, \ldots, V_n := E_n]$ — *loop continue* (B_r12)

| **hide** $G_1 : \Gamma_1, \ldots, G_n : \Gamma_n$ **in** B_0 [**endhide**] — (B_r13)

| $\Pi\,[G'_1, \ldots, G'_p]\,(E_1 \mid \&P_1, \ldots, E_n \mid \&P_n)$ — *process instantiation* (B_r14)

| **rename** — *post-renaming* (B_r15)
$G_0[(V_1^0 : T_1^0, \ldots V_{n_0}^0 : T_{n_0}^0)]$:= $G'_0[(E_1'^0, \ldots, E_{m_0}'^0)]$
...
$G_p[(V_1^p : T_1^p, \ldots V_{n_p}^p : T_{n_p}^p)]$:= $G'_p[(E_1'^p, \ldots, E_{m_p}'^p)]$
in B
endren

4.5 Abstract syntax of the minimal core language

4.5.1 Declarations

D ::= **type** T **is** T' **endtype** *type synonym* ($\mathrm{D}_m 1$)

| **type** T **is** *type declaration* ($\mathrm{D}_m 2$)
$C_1[(V_1^1 : T_1^1, ..., V_{n_1}^1 : T_{n_1}^1)]$
...
$C_p[(V_1^p : T_1^p, ..., V_{n_p}^p : T_{n_p}^p)]$
endtype $[T]$

| **channel** Γ **is** *channel declaration* ($\mathrm{D}_m 3$)
$(V_1^1 : T_1^1, ..., V_{n_1}^1 : T_{n_1}^1)$
...
$(V_1^p : T_1^p, ..., V_{n_p}^p : T_{n_p}^p)$
endchan

| **process** $\Pi[G_1 : \Gamma_1, \ldots, G_p : \Gamma_p]$ *process declaration* ($\mathrm{D}_m 4$)
(**in** | **out** $V_1 : T_1, ...,$ **in** | **out** $V_n : T_n$)
[**:noexit** | **:exit** $(T_1', ..., T_m')$] **is**
B
endproc $[\Pi]$

4.5.2 Patterns

P ::= K *constant* ($\mathrm{P}_m 1$)

| V *variable* ($\mathrm{P}_m 2$)

| **any** T ($\mathrm{P}_m 3$)

| $C(P_1, ..., P_n)$ *constructed pattern* ($\mathrm{P}_m 4$)

4.5.3 Match expressions

M ::= $E :: P$ ($\mathrm{M}_m 1$)

| M_1 **when** E ($\mathrm{M}_m 2$)

| M_1 **and** M_2 (M$_m$3)

| M_1 **or** M_2 (M$_m$4)

4.5.4 Values terms

E ::= K — *constant denotation* (E$_m$1)

| V — *value variable* (E$_m$2)

| $C(E_1, ..., E_n)$ — *constructor application* (E$_m$3)

| E_1=E_2 — *equality* (E$_m$4)

4.5.5 Behaviour expressions

B ::= $G(E_1 \mid \&P_1, \ldots, E_n \mid \&P_n)$ [[E]] — *action* (B$_m$1)

| **trap** — *trap gates* (B$_m$2)
$G_1[(V_1^1 : T_1^1, ..., V_{p_1}^1 : T_{p_1}^1)] \rightarrow B_1$
...
$G_n[(V_1^n : T_1^n, ..., V_{p_n}^n : T_{p_n}^n)] \rightarrow B_n$
in B
endtrap

| P := **any** T — *choice over values* (B$_m$3)

| **par** $G_1 : \Sigma_1, \ldots, G_p : \Sigma_p$ — *general parallel composition* (B$_m$4)
B_0 || ... || B_n
endpar

| **par** $V_1 : T_1, \ldots, V_n : T_n$ || B_0 [**endpar**] — *parallel over values* (B$_m$5)

| **case** — *general case* (B$_m$6)
$M_0 \rightarrow B_0$
...
$M_p \rightarrow B_p$
endcase

| $\mathbf{local}\ V_1 : T_1, \ldots, V_n : T_n\ \mathbf{in}\ B\ \mathbf{endloc}$ *local declaration* $(\mathrm{B}_m 7)$

| $\mathbf{hide}\ G_1 : \Gamma_1, \ldots, G_n : \Gamma_n\ \mathbf{in}\ B_0\ \mathbf{endhide}$ $(\mathrm{B}_m 8)$

| $\Pi\,[G'_1, \ldots, G'_p]\,(E_1 \mid \&P_1, \ldots, E_n \mid \&P_n)$ *process instantiation* $(\mathrm{B}_m 9)$

| **rename** *post-renaming* $(\mathrm{B}_m 10)$

$G_0[(V_1^0 : T_1^0, \ldots V_{n_0}^0 : T_{n_0}^0)]\ :=\ G'_0[(E_1'^0, \ldots, E_{m_0}'^0)]$

...

$G_p[(V_1^p : T_1^p, \ldots V_{n_p}^p : T_{n_p}^p)]\ :=\ G'_p[(E_1'^p, \ldots, E_{m_p}'^p)]$

in B

endren

4.6 Translation function from full to the reduced user language

Several constructions proposed in the full language can be translated in the reduced language using only contextual informations. For instance, using only profile informations about types, gates, channels, functions, and processes, all operators noted with $\star$ in the full abstract syntax (sub-section 4.3) will be translated in operators of reduced language (sub-section 4.4).

We present in this section the translation function.

4.6.1 Contexts

The translation function depends on informations stoked in the current *context*. We begin the presentation of its from by giving the meaning of notations used.

When A and B are sets, $\mathrm{Fin}(A)$ denotes the set of finite subsets of A, and $A \stackrel{\text{fin}}{\to} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B. The domain and range of a finite map, f, are denoted $\mathrm{Dom}(f)$ and $\mathrm{Ran}(f)$. A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$; in particular, the empty map is $\{\}$.

When A and B are sets, $A \stackrel{\text{sfin}}{\to} \mathrm{Fin}(B)$ denotes the set of *finite set-maps* from A to $\mathrm{Fin}(B)$ (the domain is a set, the range is a multi-set).

The domain *Context* is a multi-set of finite maps (for bindings of types, channels, and processes) or finite set-maps (for bindings of gates, constructors, and functions). It is built from identifiers and some constants.

A	$::=$	in	*formal parameters attributes*
	$\mid$	out	
tl	$::=$	$(T_1, \ldots, T_k)$	*list of types*
vl	$::=$	$(V_1 : T_1 : A_1, \ldots, V_k : T_k : A_k)$	*list of attributed formal parameters*
cht	$::=$	$\{vl_1, \ldots, vl_k\}$	*channel type*
gl	$::=$	$(G_1 : cht_1, \ldots, G_k : cht_k)$	*typed gate list*
$profile$	$::=$	$vl \to (T) \to uid$	*profile of constructors*
	$\mid$	$gl \to vl \to tl \to uid$	*function profile*
$\mathcal{C}$	$::=$	$T \mapsto T'$	*context of types*
	$\mid$	$C \mapsto \{profile_1, \ldots, profile_n\}$	*context of constructors*
	$\mid$	$\Gamma \mapsto cht$	*context of channels*
	$\mid$	$G \mapsto cht$	
	$\mid$	$F \mapsto \{profile_1, \ldots, profile_n\}$	*context of functions*
	$\mid$	$\Pi \mapsto gl \to vl \to tl$	*context of processes*
	$\mid$	$\{\}$	*empty context*
	$\mid$	$\mathcal{C} + \mathcal{C}$	*context composition*

uid are unique identifiers used to solve overloading of functions and constructors. In the case of functions profile, the list tl has no more than one element. If it is empty, the function type is "**void**".

Remark: The context presented above is a subset of context used for static semantics in section 4.7.

The composition of contexts for type, channel, and process contexts is a *disjunct* composition of finite maps, that is:

$$(\mathcal{C}_1 + \mathcal{C}_2)(T) = \begin{cases} \mathcal{C}_1(T) & \textbf{if } T \notin \mathrm{Dom}(\mathcal{C}_2) \\ \mathcal{C}_2(T) & \textbf{if } T \notin \mathrm{Dom}(\mathcal{C}_1) \\ \mathit{error} & \textbf{otherwise} \end{cases}$$

and similarly for channels and processes.

For gates, the composition of contexts is an *overriding* one:

$$(\mathcal{C}_1 + \mathcal{C}_2)(G) = \begin{cases} \mathcal{C}_1(G) & \textbf{if } T \notin \mathrm{Dom}(\mathcal{C}_2) \\ \mathcal{C}_2(G) & \textbf{otherwise} \end{cases}$$

The composition of contexts for constructors, and functions is defined to support overloading. So,

$$(\mathcal{C}_1 + \mathcal{C}_2)(F) = \begin{cases} \mathcal{C}_1(F) & \textbf{if } F \notin \mathrm{Dom}(\mathcal{C}_1) \\ \mathcal{C}_2(F) & \textbf{if } F \notin \mathrm{Dom}(\mathcal{C}_2) \\ \mathcal{C}_1(F) \uplus \mathcal{C}_2(F) & \textbf{otherwise} \end{cases}$$

and similarly for constructors. The function "$\uplus$" is a commutative operation defined by:

$$\begin{array}{l} \{\mathit{profile}_1, \ldots, \mathit{profile}_n\} \uplus \{\mathit{profile}'_1, \ldots, \mathit{profile}'_m\} \\ \quad \stackrel{\mathtt{def}}{=} \mathit{profile}_1 \uplus \ldots \uplus \mathit{profile}_n \uplus \\ \quad\quad \mathit{profile}'_1 \uplus \ldots \uplus \mathit{profile}'_m \end{array}$$

$$\begin{array}{l} ([gl \rightarrow](V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n) \rightarrow T_{n+1} \rightarrow \mathit{uid}) \uplus \\ ([gl' \rightarrow](V'_1 : T'_1 : A'_1, \ldots, V'_m : T'_m : A'_m) \rightarrow T'_{m+1} \rightarrow \mathit{uid}') \\ \quad \stackrel{\mathtt{def}}{=} \textbf{if } (m \neq n) \vee \\ \quad\quad ((n = m) \wedge (\forall i \in 1..m \quad (V_i = V'_i) \wedge (A_i = A'_i)) \\ \quad\quad \wedge (\exists j \in 1..(m+1) \quad T_j \neq T'_j)) \\ \quad \textbf{then} \\ \quad\quad \{([gl \rightarrow](V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n) \rightarrow T_{n+1} \rightarrow \mathit{uid}), \\ \quad\quad ([gl' \rightarrow](V'_1 : T'_1 : A'_1, \ldots, V_m : T_m : A_m \rightarrow T'_{m+1} \rightarrow \mathit{uid}')\} \\ \quad \textbf{elsif} \\ \quad\quad \textit{error: bad function overloading} \end{array}$$

The contexts for types, constructors, channels, functions, and gates are computed from declarations. The notation $\mathcal{C} \vdash D \Rightarrow \mathcal{C}'$ means "In the context $\mathcal{C}$, the declaration D gives the context $\mathcal{C}$' ". We mention that no type checking is made in the context computation.

Type declaration:

$$\begin{array}{l} \mathcal{C} \vdash \textbf{type } T \textbf{ is } T' \textbf{ endtype} \\ \quad \Rightarrow \quad (T \mapsto T') \end{array}$$

$$\mathcal{C} \vdash \left(\begin{array}{l} \textbf{type } T \textbf{ is} \\ \quad C_1(V_1^1 : T_1^1, \ldots, V_{n_1}^1 : T_{n_1}^1) \\ \quad \ldots \\ \quad C_p(V_1^p : T_1^p, \ldots, V_{n_p}^p : T_{n_p}^p) \\ \textbf{endtype} \end{array} \right)$$

$$\begin{array}{ll} \Rightarrow & (T \mapsto T) + (C_1 \mapsto vl_1 \rightarrow (T) \rightarrow uid_1) + \ldots + (C_p \mapsto vl_p \rightarrow (T) \rightarrow uid_p) \\ where & \forall i \in 1..p \quad vl_i = (V_1^i : T_1^i : in, \ldots, V_{n_i}^i : T_{n_i}^i : in) \end{array}$$

Exception declaration:

$$\mathcal{C} \vdash \left(\begin{array}{l} \textbf{exception } \Xi \textbf{ is} \\ \quad (V_1 : T_1, ..., V_n : T_n) \\ \textbf{endexc} \end{array} \right)$$
$$\Rightarrow \quad \Xi \mapsto \{(V_1 : T_1 : in, \ldots, V_n : T_n : in)\}$$

Channel declaration:

$$\mathcal{C} \vdash \left(\begin{array}{l} \textbf{channel } \Gamma \textbf{ is} \\ \quad (V_1^1 : T_1^1, \ldots, V_{n_1}^1 : T_{n_1}^1) \quad \ldots \\ \quad C_p (V_1^p : T_1^p, \ldots, V_{n_p}^p : T_{n_p}^p) \\ \textbf{endchan} \end{array} \right)$$
$$\begin{array}{ll} \Rightarrow & \Gamma \mapsto \{vl_1, \ldots, vl_p\} \\ where & \forall i \in 1..p \quad vl_i = (V_1^i : T_1^i : in, \ldots, V_{n_i}^i : T_{n_i}^i : in) \end{array}$$

Function declaration:

$$\mathcal{C} \vdash \left(\begin{array}{l} \textbf{function } F\, [X_1 : \Xi_1, ..., X_p : \Xi_p] \\ \qquad\qquad ([\textbf{in} \mid \textbf{out}]\ V_1 : T_1, ..., [\textbf{in} \mid \textbf{out}]\ V_n : T_n) \\ \qquad [:T] \textbf{ is} \\ \quad E \\ \textbf{endfunc} \end{array} \right)$$
$$\Rightarrow \qquad F \mapsto gl \rightarrow vl \rightarrow tl \rightarrow newuid$$

$where$

$$gl = (X_1 : \mathcal{C}(X_1), \ldots, X_p : \mathcal{C}(\Xi_p))$$
$$vl = (V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n)$$
$$\forall i \in 1..n \quad A_i = \begin{cases} out & \textbf{if out} V_i : T_i \\ in & \textbf{otherwise} \end{cases}$$
$$tl = \begin{cases} (T) & \textbf{if } :T \\ () & \textbf{otherwise} \end{cases}$$

Process declaration:

$$\mathcal{C} \vdash \left(\begin{array}{l} \textbf{process } \Pi\,[G_1\!:\!\Gamma_1,\ldots,G_p\!:\!\Gamma_p] \\ \qquad\qquad ([\textbf{in} \mid \textbf{out}]\ V_1\!:\!T_1,\ldots,[\textbf{in} \mid \textbf{out}]\ V_n\!:\!T_n) \\ \qquad\quad [:\textbf{noexit} \mid \textbf{exit}\ (T_1',\ldots,T_m')]\ \textbf{is} \\ \quad B \\ \textbf{endproc} \end{array} \right)$$

$$\Rightarrow \qquad P \mapsto gl \rightarrow vl \rightarrow tl$$

where

$$gl = (G_1 : \mathcal{C}(\Gamma_1), \ldots, X_p : \mathcal{C}(\Gamma_p))$$

$$vl = (V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n)$$

$$\forall i \in 1..n \quad A_i = \begin{cases} out & \textbf{if out} V_i\!:\!T_i \\ in & \textbf{otherwise} \end{cases}$$

$$tl = \begin{cases} (T_1', \ldots, T_m') & \textbf{if exit}(T_1', \ldots, T_m') \\ () & \textbf{otherwise} \end{cases}$$

The result context of each declaration is composed with the contexts resulting from other declarations at the module level.

4.6.2 Translation function

The translation function is a contextual-dependent morphism. It translates each derived form (marked by $\star$ in full grammar) in the reduced user language.

The table below indicate the notations used for classes of phrases of two grammars:

full user language		*reduced user language*	
symbol	*section*	*symbol*	*section*
$\mathcal{D}_f$	4.3.1	$\mathcal{D}_r$	4.4.1
$\mathcal{P}_f$	4.3.2	$\mathcal{P}_r$	4.4.2
$\mathcal{M}_f$	4.3.3	$\mathcal{M}_r$	4.4.3
$\mathcal{E}_f$	4.3.4	$\mathcal{E}_r$	4.4.4
$\mathcal{B}_f$	4.3.5	$\mathcal{B}_r$	4.4.5

The translation morphism $[[.,.]]$ from full user language to reduced user language has the profile

$$\begin{array}{lll} [[.]]^{\cdot} & : & (Context \rightarrow \mathcal{D}_f \rightarrow \mathcal{D}_r) \cup \\ & & (Context \rightarrow \mathcal{P}_f \rightarrow \mathcal{P}_r) \cup \\ & & (Context \rightarrow \mathcal{M}_f \rightarrow \mathcal{M}_r) \cup \\ & & (Context \rightarrow \mathcal{E}_f \rightarrow \mathcal{E}_r) \cup \\ & & (Context \rightarrow \mathcal{B}_f \rightarrow \mathcal{B}_r) \end{array}$$

4.6.3 Translation of declarations

The translation morphism for declarations is an identity morphism for type, exception, and channel declarations.

Function declaration:

$$\left[\left[\mathcal{C},\ \begin{pmatrix} \textbf{function } F\,[X_1\!:\!\Xi_1,\ldots,X_p\!:\!\Xi_p] \\ (\ldots V_i\!:\!T_i\ldots) \\ [:T]\ \textbf{is} \\ E \\ \textbf{endfunc} \end{pmatrix}\right]\right] \stackrel{\texttt{def}}{=} \begin{pmatrix} \textbf{function } F\,[X_1\!:\!\Xi_1,\ldots,X_p\!:\!\Xi_p] \\ (\ldots \textbf{in } V_i\!:\!T_i,\ldots) \\ [:T]\ \textbf{is} \\ [[\mathcal{C}',\ E\,]] \\ \textbf{endfunc} \end{pmatrix}$$

where $\mathcal{C}' = \mathcal{C} + (X_1 \Rightarrow \Xi_1) + \ldots + (X_p \Rightarrow \Xi_p)$. So, attribute "*in*" is given by default, and declared exceptions are added to evaluate the body of function.

Process declaration: similar to function declaration.

4.6.4 Translation of patterns

The translation morphism for patterns is an identity morphism for all patterns except constructed pattern.

Constructed patterns: There are three phases: (1) reordering of actual parameters iff formal parameters are given, (2) replacing "**...**" with the missing parameters, and (3) removing formal parameter annotations. Error are signaled if "**...**" are given for positional call, or named actual parameters are merged with positional parameters. Thus,

$$[[\mathcal{C},\ C(P_1,\ldots,P_n,\ \textbf{...})\ ]] \stackrel{\texttt{def}}{=} error$$

$$[[\mathcal{C},\ C(\ldots P_i,\ldots V_j := P_j\ \ldots)\ ]] \stackrel{\texttt{def}}{=} error$$

$$[[\mathcal{C},\ C(V_1 := P_1,\ldots,V_n := P_n)\ ]]$$
$$\stackrel{\texttt{def}}{=} \begin{cases} C(P'_1,\ldots,P'_n) \\ \quad \textbf{if } (\exists! \rho \in P(n))\ (\exists((V'_1 : T_1 : in,\ldots,V'_n : T_n : in) \rightarrow T) \in \mathcal{C}(C)) \\ \qquad s.t.\ (V_{\rho(i)} = V'_i) \\ error \\ \quad \textbf{otherwise} \end{cases}$$
$$where\ \forall i \in 1..n \quad P'_i = [[\mathcal{C},\ P_{\rho(i)}\ ]]$$

$$[[\mathcal{C},\ C(V_1 := P_1,\ldots,V_n := P_n,\ \textbf{...})\ ]]$$
$$\stackrel{\texttt{def}}{=} \begin{cases} C(P'_1,\ldots,P'_m) \\ \quad \textbf{if } (\exists m)\ (\exists \rho \in P(m)\ (\exists!((V'_1 : T_1 : in,\ldots,V'_m : T_m : in) \rightarrow T) \in \mathcal{C}(C)) \\ \qquad s.t.\ (m > n)\ \wedge\ (\forall i \in 1..n \quad V_{\rho(i)} = V'_i) \\ error \\ \quad \textbf{otherwise} \end{cases}$$
$$where\ \forall i \in 1..m \quad P'_i = \begin{cases} [[\mathcal{C},\ P_{\rho(i)}\ ]] & \textbf{if } i \in 1..n \\ V'_i & \textbf{otherwise} \end{cases}$$

($P(n)$ is the set of permutations of $1..n$.)

4.6.5 Translation of match expressions

$$[[\mathcal{C},\ E\texttt{::}P\ ]] \stackrel{\texttt{def}}{=} [[\mathcal{C},\ E\ ]]\texttt{::}[[\mathcal{C},\ P\ ]]$$

For the other match expressions, the translation morphism is an identity morphism.

4.6.6 Translation of value expressions

We present in the following only the cases for which the translation morphism is not an identity. The translation function raises error if:

- "$\texttt{...}$" are present in positional list of actual parameters, for instance $C(E_1,\ldots,E_n,\texttt{...})$ is forbidden, and similarly for exceptions lists;
- named (e.g. "V:=E") and positional (e.g. "E") styles for actual parameter lists (values and exceptions) are merged.

In the following we don't present this error cases.

Constructor application:

$$
\begin{array}{l}
[[\mathcal{C},\ C(V_1\texttt{:=}E_1,\ldots,V_n\texttt{:=}E_n)\ ]] \\
\stackrel{\texttt{def}}{=} \left\{ \begin{array}{l}
C(E'_1,\ldots,E'_n) \\
\quad \textbf{if } (\exists! \rho \in P(n))\ (\exists((V'_1 : T_1 : in,\ldots,V'_n : T_n : in) \to T) \in \mathcal{C}(C)) \\
\qquad s.t.\ (V_{\rho(i)} = V'_i) \\
error \\
\quad \textbf{otherwise}
\end{array} \right. \\
\quad where\ \forall i \in 1..n \quad E'_i = [[\mathcal{C},\ E_{\rho(i)}\ ]]
\end{array}
$$

$$
\begin{array}{l}
[[\mathcal{C},\ C(V_1\texttt{:=}E_1,\ldots,V_n\texttt{:=}E_n,\texttt{...})\ ]] \\
\stackrel{\texttt{def}}{=} \left\{ \begin{array}{l}
C(E'_1,\ldots,E'_m) \\
\quad \textbf{if } (\exists m)\ (\exists \rho \in P(m)\ (\exists!((V'_1 : T_1 : in,\ldots,V'_m : T_m : in) \to T) \in \mathcal{C}(C)) \\
\qquad s.t.\ (m > n)\ \wedge\ (\forall i \in 1..n \quad V_{\rho(i)} = V'_i) \\
error \\
\quad \textbf{otherwise}
\end{array} \right. \\
\quad where\ \forall i \in 1..m \\
\quad E'_i = \left\{ \begin{array}{ll}
[[\mathcal{C},\ P_{\rho(i)}\ ]] & \textbf{if } i \in 1..n \\
V'_i & \textbf{otherwise}
\end{array} \right.
\end{array}
$$

Function application:

$$
\begin{array}{l}
[[\mathcal{C},\ F\texttt{[}X_1,\ldots,X_p\texttt{]}(V_1\texttt{:=}E_1 \mid V_1\texttt{=:\&}P_1,\ldots,V_n\texttt{:=}E_n \mid V_n\texttt{=:\&}P_n)\ ]] \\
\stackrel{\texttt{def}}{=} \left\{ \begin{array}{l}
F\texttt{[}X_1,\ldots,X_p\texttt{]}(E'_1 \mid \texttt{\&}P'_1,\ldots,E'_n \mid \texttt{\&}P'_n) \\
\quad \textbf{if } (\exists! \rho \in P(n))\ (\exists(gl \to (V'_1 : T_1 : A'_1,\ldots,V'_n : T_n : A'_n) \to T) \in \mathcal{C}(F)) \\
\qquad s.t.\ (length\ (gl) = p)\ \wedge \\
\qquad\quad (\forall i \in 1..n \quad (V_{\rho(i)} = V'_i)\ \wedge\ (A_{\rho(i)} = A'_i)) \\
error \\
\quad \textbf{otherwise}
\end{array} \right.
\end{array}
$$

$$
\begin{aligned}
& where \forall i \in 1..n \\
& E'_i \mid P'_i = [[\mathcal{C},\ E_{\rho(i)}\ ]] \mid [[\mathcal{C},\ P_{\rho(i)}\ ]] \\
& A_i = \begin{cases} out & \textbf{if } \&P_i \\ in & \textbf{otherwise} \end{cases}
\end{aligned}
$$

$$
[[\mathcal{C},\ F\,[X_1,\ldots,X_p]\,(V_1 := E_1 \mid V_1 =: \&P_1, \ldots, V_n := E_n \mid V_n =: \&P_n, \ldots)\ ]]
$$

$$
\stackrel{\text{def}}{=} \begin{cases}
F\,[X_1,\ldots,X_p]\,(E'_1 \mid \&P'_1, \ldots, E'_m \mid \&P'_m) \\
\quad \textbf{if } (\exists m)\ (\exists! \rho \in P(m))\ (\exists (gl \rightarrow (V'_1 : T_1 : A'_1, \ldots, V'_m : T_m : A'_m) \rightarrow T) \in \mathcal{C}(F)) \\
\quad\quad s.t.\ (length\ (gl) = p)\ \wedge\ (m > n)\ \wedge \\
\quad\quad\quad (\forall i \in 1..n \quad (V_{\rho(i)} = V'_i)\ \wedge\ (A_{\rho(i)} = A'_i)) \\
error \\
\quad \textbf{otherwise}
\end{cases}
$$

$$
\begin{aligned}
& where \forall i \in 1..m \\
& E'_i \mid P'_i = \begin{cases} [[\mathcal{C},\ E_{\rho(i)}\ ]] \mid [[\mathcal{C},\ P_{\rho(i)}\ ]] & \textbf{if } i \in 1..n \\ V'_i & \textbf{otherwise} \end{cases} \\
& A_i = \begin{cases} out & \textbf{if } \&P_i \\ in & \textbf{otherwise} \end{cases}
\end{aligned}
$$

$$
[[\mathcal{C},\ F\,[X'_1 := X_1, \ldots, X'_p := X_p]\,(V_1 := E_1 \mid V_1 =: \&P_1, \ldots, V_n := E_n \mid V_n =: \&P_n)\ ]]
$$

$$
\stackrel{\text{def}}{=} \begin{cases}
F\,[X'''_1, \ldots, X'''_p]\ (E'_1 \mid \&P'_1, \ldots, E'_n \mid \&P'_n) \\
\quad \textbf{if } (\exists! \rho \in P(n))\ (\exists! \rho' \in P(p)) \\
\quad\quad (\exists ((X''_1 : \Xi_1, \ldots, X''_p : \Xi_p) \rightarrow (V'_1 : T_1 : A'_1, \ldots, V'_n : T_n : A'_n) \rightarrow T) \in \mathcal{C}(F)) \\
\quad\quad s.t.\ (length\ (gl) = p)\ \wedge \\
\quad\quad\quad (\forall i \in 1..n \quad (V_{\rho(i)} = V'_i)\ \wedge\ (A_{\rho(i)} = A'_i)) \\
error \\
\quad \textbf{otherwise}
\end{cases}
$$

$$
\begin{aligned}
& where \forall i \in 1..n\ \forall j \in 1..p \\
& E'_i \mid P'_i = [[\mathcal{C},\ E_{\rho(i)}\ ]] \mid [[\mathcal{C},\ P_{\rho(i)}\ ]] \\
& A_i = \begin{cases} out & \textbf{if } \&P_i \\ in & \textbf{otherwise} \end{cases} \\
& X'''_j = X_{\rho'(j)}
\end{aligned}
$$

$$
[[\mathcal{C},\ F\,[X'_1 := X_1, \ldots, X'_p := X_p, \ldots]\,(V_1 := E_1 \mid P_1, \ldots, V_n = E_n \mid P_n, \ldots)\ ]]
$$

$$
\stackrel{\text{def}}{=} \begin{cases}
F\,[X'''_1, \ldots, X'''_r]\ (E'_1 \mid \&P'_1, \ldots, E'_m \mid \&P'_m) \\
\quad \textbf{if } (\exists m)\ (\exists r) \\
\quad\quad (\exists! \rho \in P(m))\ (\exists! \rho' \in P(r)) \\
\quad\quad (\exists ((X''_1 : \Xi_1, \ldots, X''_r : \Xi_r) \rightarrow (V'_1 : T_1 : A'_1, \ldots, V'_m : T_m : A'_m) \rightarrow T) \in \mathcal{C}(F)) \\
\quad\quad s.t.\ (m > n)\ \wedge\ (r > p)\ \wedge \\
\quad\quad\quad (\forall i \in 1..n \quad (V_{\rho(i)} = V'_i)\ \wedge (A_{\rho(i)} = A'_i))\ \wedge \\
\quad\quad\quad (\forall j \in 1..p \quad X''_j = X'_{\rho'(j)}) \\
error \\
\quad \textbf{otherwise}
\end{cases}
$$

$$
\begin{aligned}
& where \forall i \in 1..m\ \forall j \in 1..r \\
& E'_i \mid P'_i = \begin{cases} [[\mathcal{C},\ E_{\rho(i)}\ ]] \mid [[\mathcal{C},\ P_{\rho(i)}\ ]] & \textbf{if } i \in 1..n \\ V'_i & \textbf{otherwise} \end{cases}
\end{aligned}
$$

$$A_i = \begin{cases} out & \textbf{if } \&P_i \\ in & \textbf{otherwise} \end{cases}$$

$$X_j''' = X_{\rho'(j)}$$

The other 2 cases are obvious.

Otherwise clause:

$$[[\mathcal{C},\ \textbf{otherwise}\ E_{n+1}\ ]] \stackrel{\text{def}}{=} \textbf{true}::\textbf{true} \rightarrow [[\mathcal{C},\ E_{n+1}\ ]]$$

Simple case expression:

$$\left[\left[\mathcal{C}, \begin{pmatrix} \textbf{case}\ E'\ \textbf{in} \\ \quad P_1^0, ..., P_{n_0}^0 \\ \qquad \textbf{when}\ E_0 \rightarrow E_0' \\ \quad ... \\ \quad P_1^p, ..., P_{n_p}^p \\ \qquad \textbf{when}\ E_p \rightarrow E_p' \\ \quad \textbf{otherwise}\ E_{p+1}' \\ \textbf{endcase} \end{pmatrix}\right]\right] \stackrel{\text{def}}{=} \begin{pmatrix} \textbf{case}\ [[\mathcal{C},\ E'\ ]]::(X) \rightarrow \\ \quad \textbf{case} \\ \qquad X::[[\mathcal{C},\ P_1^0\ ]]\ \textbf{or}\ ...\ \textbf{or}\ X::[[\mathcal{C},\ P_{n_0}^0\ ]] \\ \qquad\quad \textbf{when}\ [[\mathcal{C},\ E_0\ ]] \rightarrow [[\mathcal{C},\ E_0'\ ]] \\ \qquad ... \\ \qquad X::[[\mathcal{C},\ P_1^p\ ]]\ \textbf{or}\ ...\ \textbf{or}\ X::[[\mathcal{C},\ P_{n_p}^p\ ]] \\ \qquad\quad \textbf{when}\ [[\mathcal{C},\ E_p\ ]] \rightarrow [[\mathcal{C},\ E_p'\ ]] \\ \qquad \textbf{true}::\textbf{true} \rightarrow [[\mathcal{C},\ E_{p+1}'\ ]] \\ \quad \textbf{endcase} \\ \textbf{endcase} \end{pmatrix}$$

Match expression:

$$[[\mathcal{C},\ E\ \textbf{match}\ P\ ]] \stackrel{\text{def}}{=} \begin{pmatrix} \textbf{case} \\ \quad [[\mathcal{C},\ E_0\ ]]::[[\mathcal{C},\ P\ ]] \rightarrow \textbf{true} \\ \quad \textbf{true}::\textbf{true} \rightarrow \textbf{false} \\ \textbf{endcase} \end{pmatrix}$$

Conditional expression:

$$\left[\left[\mathcal{C}, \begin{pmatrix} \textbf{if}\ E_0\ \textbf{then}\ E_0' \\ \textbf{elsif}\ E_1\ \textbf{then}\ E_1' \\ ... \\ \textbf{elsif}\ E_n\ \textbf{then}\ E_n' \\ \textbf{else}\ E_{n+1}' \\ \textbf{endif} \end{pmatrix}\right]\right] \stackrel{\text{def}}{=} \begin{pmatrix} \textbf{case} \\ \quad [[\mathcal{C},\ E_0\ ]]::\textbf{true} \rightarrow [[\mathcal{C},\ E_0'\ ]] \\ \quad [[\mathcal{C},\ E_1\ ]]::\textbf{true} \rightarrow [[\mathcal{C},\ E_1'\ ]] \\ \quad ... \\ \quad [[\mathcal{C},\ E_n\ ]]::\textbf{true} \rightarrow [[\mathcal{C},\ E_n'\ ]] \\ \quad \textbf{true}::\textbf{true} \rightarrow [[\mathcal{C},\ E_{n+1}'\ ]] \\ \textbf{endcase} \end{pmatrix}$$

Logical expression:

$$[[\mathcal{C},\ E_0\ \textbf{andthen}\ E_1\ ]] \stackrel{\text{def}}{=} \begin{pmatrix} \textbf{case} \\ \quad [[\mathcal{C},\ E_0\ ]]::\textbf{true} \rightarrow [[\mathcal{C},\ E_1\ ]] \\ \quad [[\mathcal{C},\ \textbf{otherwise} \rightarrow \textbf{false}\ ]] \\ \textbf{endcase} \end{pmatrix}$$

$$[[\mathcal{C},\ E_0\ \textbf{orelse}\ E_1\ ]] \ \stackrel{\texttt{def}}{=}\ \left(\begin{array}{l}\textbf{case}\\ \quad [[\mathcal{C},\ E_0\ ]]\texttt{::}\textbf{true} \rightarrow \textbf{true}\\ \quad \textbf{true}\texttt{::}\textbf{true} \rightarrow [[\mathcal{C},\ E_1\ ]]\\ \textbf{endcase}\end{array}\right)$$

Exception raise:

$$\begin{array}{l}[[\mathcal{C},\ \textbf{raise}\ X(V_1\texttt{:=}E_1,\ldots,V_n\texttt{=}E_n)\ ]]\\ \stackrel{\texttt{def}}{=} \left\{\begin{array}{ll}\textbf{raise}\ X(E_1',\ldots,E_n') & \textbf{if}\ (\mathcal{C}(X) = \{(V_1':T_1:in,\ldots,V_n':T_n:in)\}) \wedge\\ & \quad (\exists!\rho \in P(n) \quad s.t.\\ & \quad\quad (\forall i \in 1..n \quad (V_{\rho(i)} = V_i'))\ )\\ error & \textbf{otherwise}\end{array}\right.\\ \quad where \forall i \in 1..n E_i' = [[\mathcal{C},\ E_{\rho(i)}\ ]]\end{array}$$

Trap exceptions:

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l}\textbf{trap}\\ \quad X_1[(V_1^1\texttt{:}T_1^1,\ldots,V_{p_1}^1\texttt{:}T_{p_1}^1)] \rightarrow E_1\\ \quad \ldots\\ \quad X_n[(V_1^n\texttt{:}T_1^n,\ldots,V_{p_n}^n\texttt{:}T_{p_n}^n)] \rightarrow E_n\\ \textbf{in}\ E\\ \textbf{endtrap}\end{array}\right)\right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{trap}\\ \quad X_1[(V_1^1\texttt{:}T_1^1,\ldots,V_{p_1}^1\texttt{:}T_{p_1}^1)]\\ \quad\quad \rightarrow [[\mathcal{C},\ E_1\ ]]\\ \quad \ldots\\ \quad X_n[(V_1^n\texttt{:}T_1^n,\ldots,V_{p_n}^n\texttt{:}T_{p_n}^n)]\\ \quad\quad \rightarrow [[\mathcal{C},\ E_n\ ]]\\ \textbf{in}\ [[\mathcal{C}',\ E\ ]]\\ \textbf{endtrap}\end{array}\right)$$

where $\mathcal{C}' = \mathcal{C} + (X_1 \mapsto \{(V_1^1 : T_1^1 : in, \ldots, V_{p_1}^1 : T_{p_1}^1 : in)\}) + \ldots (X_n \mapsto \{V_n^1\texttt{:}T_n^1, \ldots, V_{p_n}^n\texttt{:}T_{p_n}^n\})$.

Non-equality expression:

$$[[\mathcal{C},\ E_0\ \texttt{<>}\ E_1\ ]] \stackrel{\texttt{def}}{=} \textbf{not}([[\mathcal{C},\ E_0\ ]]\texttt{=}[[\mathcal{C},\ E_1\ ]])$$

where "**not**" is the predefined function over booleans.

Let expression:

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l}\textbf{let}\ V_1\texttt{:}T_1\texttt{:=}E_1,\ldots,\\ \quad\quad V_n\texttt{:}T_n\texttt{:=}E_n\\ \textbf{in}\ E\end{array}\right)\right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{local}\ V_1\texttt{:}T_1,\ldots,V_n\texttt{:}T_n\ \textbf{in}\\ \quad \textbf{case}\\ \quad\quad [[\mathcal{C},\ E_1\ ]]\texttt{::}\&V_1\ \textbf{and}\ \ldots\ \textbf{and}\\ \quad\quad [[\mathcal{C},\ E_n\ ]]\texttt{::}\&V_n \rightarrow [[\mathcal{C},\ E\ ]]\\ \quad \textbf{endcase}\\ \textbf{endloc}\end{array}\right)$$

4.6.7 Translation of behaviour expressions

We present in the following only the cases for which the translation morphism is not an identity. The translation function raises error if:

- "**...**" are present in positional list of actual parameters, i.e. $C(E_1, \ldots, E_n, \textbf{...})$ is forbidden, and similarly for gate lists and gate offers;

- named (e.g. "V:=E") and positional (e.g. "E") styles for actual parameter lists (both gates and values) are merged.

In the following we don't present this error cases.

Action:

$$[[\mathcal{C},\ G(V_1\texttt{:=!}E_1 \mid V_1\texttt{=:\&}P_1 \ldots V_n\texttt{:=!}E_n \mid V_n\texttt{=:\&}P_n)\texttt{[}E_{\text{bool}}\texttt{]}\ ]]$$

$$\stackrel{\texttt{def}}{=} \begin{cases} G(O_1 \ldots O_n)\texttt{[}[[\mathcal{C},\ E_{\text{bool}}\ ]]\texttt{]} \\ \quad \textbf{if } (\exists \rho \in P(n)) \\ \qquad (\exists (V_1' : T_1 : in, \ldots, V_n' : T_n : in) \in \mathcal{C}(G)) \\ \qquad s.t. (\forall i \in 1..n \quad V_i' = V_{\rho(i)}) \\ error \\ \quad \textbf{otherwise} \end{cases}$$

$$where$$

$$\forall i \in 1..n \quad O_i = [[\mathcal{C},\ E_{\rho(i)}\ ]] \mid \texttt{\&}[[\mathcal{C},\ P_{\rho(i)}\ ]]$$

$$[[\mathcal{C},\ G(V_1\texttt{:=!}E_1 \mid V_1\texttt{=:\&}P_1, \ldots, V_n\texttt{:=!}E_n \mid V_n\texttt{=:\&}P_n, \texttt{...})\texttt{[}E_{\text{bool}}\texttt{]}\ ]]$$

$$\stackrel{\texttt{def}}{=} \begin{cases} G(O_1, \ldots, O_m)\texttt{[}[[\mathcal{C},\ E_{\text{bool}}\ ]]\texttt{]} \\ \quad \textbf{if } (\exists m)\ (\exists ! \rho \in P(m)) \\ \qquad (\exists (V_1' : T_1 : in, \ldots, V_m' : T_m : in) \in \mathcal{C}(G)) \\ \qquad s.t.\ (m > n)\ \wedge\ (\forall i \in 1..n \quad (V_i' = V_{\rho(i)})) \\ error \\ \quad \textbf{otherwise} \end{cases}$$

$$where$$

$$\forall i \in 1..m \quad O_i = \begin{cases} [[\mathcal{C},\ E_{\rho(i)}\ ]] \mid \texttt{\&}[[\mathcal{C},\ P_{\rho(i)}\ ]] & \textbf{if } i \in 1..n \\ V_i & \textbf{otherwise} \end{cases}$$

Exception raise:

$$[[\mathcal{C},\ \textbf{raise}\ G(V_1\texttt{:=}E_1, \ldots, V_n\texttt{:=}E_n)\ ]]$$

$$\stackrel{\texttt{def}}{=} \begin{cases} G(E_1' \ldots E_n') \\ \quad \textbf{if } (Ctx(G) = \{(V_1' : T_1 : in, \ldots, V_n' : T_n : in)\}\ \wedge\ (\exists \rho \in P(n)) \\ \qquad s.t. (\forall i \in 1..n \quad V_i' = V_{\rho(i)}) \\ error \\ \quad \textbf{otherwise} \end{cases}$$

$$where\ \forall i \in 1..n E_i' = [[\mathcal{C},\ E_{\rho(i)}\ ]]$$

$$[[\mathcal{C},\ \textbf{raise}\ G(V_1\texttt{:=}E_1, \ldots, V_n\texttt{:=}E_n, \texttt{...})\ ]]$$

$$\stackrel{\texttt{def}}{=} \begin{cases} G(E_1', \ldots, E_m') \\ \quad \textbf{if } (Ctx(G) = \{(V_1' : T_1 : in, \ldots, V_m' : T_m : in)\}\ \wedge\ (m > n) \wedge \\ \qquad ((\exists \rho \in P(m)) \quad (\forall i \in 1..n \quad (V_i' = V_{\rho(i)}))) \\ error \\ \quad \textbf{otherwise} \end{cases}$$

$$where$$

$$\forall i \in 1..m \quad E_i = \begin{cases} [[\mathcal{C},\ E_{\rho(i)}]] & \textbf{if } i \in 1..n \\ V_i & \textbf{otherwise} \end{cases}$$

Deadlock:

$$[[\mathcal{C},\ \textbf{stop}]] \stackrel{\texttt{def}}{=} \Pi_{\textbf{stop}}$$
$$\textit{where } \textbf{proces } \Pi_{\textbf{stop}} \textbf{:noexit is } \Pi_{\textbf{stop}} \textbf{ endproc}$$

Successful termination:

$$[[\mathcal{C},\ \textbf{exit}]] \stackrel{\texttt{def}}{=} \delta$$

$$[[\mathcal{C},\ \textbf{exit}\ (\ldots E_i \ldots \textbf{any}\ T_j \ldots)]]$$
$$\stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{local}\ \ldots X_j\!:\!T_j \ldots\ \textbf{in} \\ \qquad \ldots \\ \qquad X_j\ \texttt{:=}\ \textbf{any}\ T_j \\ \qquad \ldots \\ \qquad \delta(\ldots [[\mathcal{C},\ E_i]] \ldots X_j \ldots) \\ \textbf{endloc} \end{array} \right)$$

Trap gates:

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l} \textbf{trap} \\ \quad G_1[(V_1^1\!:\!T_1^1, ..., V_{p_1}^1\!:\!T_{p_1}^1)] \\ \qquad \rightarrow B_1 \\ \quad \ldots \\ \quad G_n[(V_1^n\!:\!T_1^n, ..., V_{p_n}^n\!:\!T_{p_n}^n)] \\ \qquad \rightarrow B_n \\ \textbf{in}\ B \\ \textbf{endtrap} \end{array} \right) \right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \quad G_1[(V_1^1\!:\!T_1^1, ..., V_{p_1}^1\!:\!T_{p_1}^1)] \\ \qquad \rightarrow [[\mathcal{C},\ B_1]] \\ \quad \ldots \\ \quad G_n[(V_1^n\!:\!T_1^n, ..., V_{p_n}^n\!:\!T_{p_n}^n)] \\ \qquad \rightarrow [[\mathcal{C},\ B_n]] \\ \textbf{in}\ [[\mathcal{C}',\ B]] \\ \textbf{endtrap} \end{array} \right)$$

where $\mathcal{C}' \stackrel{\texttt{def}}{=} \mathcal{C} + (G_1 \mapsto \{(V_1^1 : T_1^1 : in, ..., V_{p_1}^1 : T_{p_1}^1 : in)\}) + \ldots + (G_n \mapsto \{V_n^1\!:\!T_n^1, ..., V_{p_n}^n\!:\!T_{p_n}^n\})$.
Different sequential compositions:

$$[[\mathcal{C},\ \textbf{seq}\ B_0\ ;\ \ldots\ ;\ B_n \textbf{endseq}]]$$
$$\stackrel{\texttt{def}}{=} (\ldots([[\mathcal{C},\ B_0]];[[\mathcal{C},\ B_1]])\ldots\ ;\ [[\mathcal{C},\ B_n]])$$

$$[[\mathcal{C},\ G\ \texttt{!}E_1 \mid \texttt{?}V_1\!:\!T_1 \ldots \texttt{!}E_n \mid \texttt{?}V_n\!:\!T_n\ \texttt{[}E_{\text{bool}}\texttt{]}\ ;\ B]]$$
$$\stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{local}\ \ldots V_j\!:\!T_j \ldots\ \textbf{in} \\ \qquad G(E_1 \mid \texttt{\&}V_1, \ldots, E_n \mid \texttt{\&}V_n)\ \texttt{[}[[\mathcal{C},\ E_{\text{bool}}]]\texttt{]}\ ;\ B \\ \textbf{endloc} \end{array} \right)$$

$$[[\mathcal{C},\ B_1\ \texttt{>>}\ \textbf{accept}\ V_1\!:\!T_1 \ldots V_n\!:\!T_n\ \textbf{in}\ B_2]]$$

$$\stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \quad \delta(V_1:T_1,\ldots,V_n:T_n) \\ \qquad \rightarrow \mathbf{i} \ ; \ [[\mathcal{C},\ B_2\,]] \\ \textbf{in}\ [[\mathcal{C},\ B_1\,]] \\ \textbf{endtrap} \end{array} \right)$$

Non-deterministic choice:

$$[[\mathcal{C},\ \textbf{alt}\ B_1\, [\,]\ \ldots\ [\,]\, B_n\ \textbf{endalt}\,]]$$

$$\stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \quad G(V:\textbf{enum}(1,n)) \rightarrow \\ \qquad\qquad \textbf{case} \\ \qquad\qquad\quad V::1 \rightarrow B_1 \\ \qquad\qquad\quad \ldots \\ \qquad\qquad\quad V::n \rightarrow B_n \\ \qquad\qquad \textbf{endcase} \\ \textbf{in} \\ \quad V\ := \textbf{any}\ \textbf{enum}(1,n); G(V) \\ \textbf{endtrap} \end{array} \right)$$

where **enum**$(1..n)$ *is a library type*

$$[[\mathcal{C},\ B_1\, [\,]\, B_2\,]]$$

$$\stackrel{\text{def}}{=} \ [[\mathcal{C},\ \textbf{alt}\ B_1\, [\,]\, B_2\ \textbf{endalt}\,]]$$

$$[[\mathcal{C},\ \textbf{choice}\ V_1:T_1,\ldots,V_n:T_n\ [\,]\ B_0\textbf{endch}\,]]$$

$$\stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{local}\ V_1:T_1,\ldots,V_n:T_n\ \textbf{in} \\ \quad V_1\ := \textbf{any}\ T_1;\ldots;V_n\ := \textbf{any}\ T_n; [[\mathcal{C},\ B_0\,]] \\ \textbf{endlo} \end{array} \right)$$

Parallel composition:

$$\left[\left[\mathcal{C}, \left(\begin{array}{l} \textbf{par} \\ \quad [G_1^0,\ldots,G_{p_0}^0]\ \textbf{for}\ B_0 \\ ||\ \ldots \\ ||\ [G_0^n,\ldots,G_{p_n}^n]\ \textbf{for}\ B_n \\ \textbf{endpar} \end{array} \right) \right]\right] \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{par}\ common(\cup_{b=0}^{b=n}(\cup_{i=0}^{i=p_b} G_i^b)) \\ \quad [[\mathcal{C},\ B_0\,]]\ ||\ \ldots\ ||\ [[\mathcal{C},\ B_n\,]] \\ \textbf{endpar} \end{array} \right)$$

where

$$common(\cup_{b=0}^{b=n}(\cup_{i=0}^{i=p_b} G_i^b)) \stackrel{\text{def}}{=} \{G:\Sigma \mid (\forall b \in \Sigma)(\exists i \in 0..p_b)\ G = G_i^b\}$$

$$\left[\left[\mathcal{C}, \left(\begin{array}{l} \textbf{par sync}\ G_1\#n_1,\ldots,G_p\#n_p \\ \quad B_0\ ||\ \ldots\ ||\ B_n \\ \textbf{endpar} \end{array} \right) \right]\right] \ \stackrel{\text{def}}{=} \ \left(\begin{array}{l} \textbf{par}\ G_1:\Sigma_{0..n}^{n_1},\ldots,G_p:\Sigma_{0..n}^{n_p} \\ \quad [[\mathcal{C},\ B_0\,]]\,||\ \ldots\,||\ [[\mathcal{C},\ B_n\,]] \\ \textbf{endpar} \end{array} \right)$$

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l}\textbf{par sync all}\\ \quad B_0\ \texttt{||}\ \ldots\ \texttt{||}\ B_n\\ \textbf{endpar}\end{array}\right)\right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{par}\ \{G\!:\!\{0..n\}\mid G\in\mathrm{Gat}\}\\ \quad [[\mathcal{C},\ B_0\ ]]\texttt{||}\ \ldots\texttt{||}\ [[\mathcal{C},\ B_n\ ]]\\ \textbf{endpar}\end{array}\right)$$

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l}\textbf{par sync none}\\ \quad B_0\ \texttt{||}\ \ldots\ \texttt{||}\ B_n\\ \\ \textbf{endpar}\end{array}\right)\right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{par}\\ \quad [[\mathcal{C},\ B_0\ ]]\texttt{||}\ \ldots\texttt{||}\ [[\mathcal{C},\ B_n\ ]]\\ \textbf{endpar}\end{array}\right)$$

$$[[\mathcal{C},\ B_1\,|[G_1\ldots G_n]|\,B_2\ ]] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{par}\ G_1\!:\!\{1,2\},\ldots,G_n\!:\!\{1,2\}\\ \quad [[\mathcal{C},\ B_1\ ]]\ \texttt{||}\ [[\mathcal{C},\ B_2\ ]]\\ \textbf{endpar}\end{array}\right)$$

$$[[\mathcal{C},\ B_1\ \texttt{||}\ B_2\ ]] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{par}\ \{G\!:\!\{1,2\}\mid G\in\mathrm{Gat}\}\\ \quad [[\mathcal{C},\ B_1\ ]]\ \texttt{||}\ [[\mathcal{C},\ B_2\ ]]\\ \textbf{endpar}\end{array}\right)$$

$$[[\mathcal{C},\ B_1\ \texttt{|||}\ B_2\ ]] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{par}\\ \quad [[\mathcal{C},\ B_1\ ]]\ \texttt{||}\ [[\mathcal{C},\ B_2\ ]]\\ \textbf{endpar}\end{array}\right)$$

Otherwise clause:

$$[[\mathcal{C},\ \textbf{otherwise}\ B_{p+1}\ ]] \stackrel{\texttt{def}}{=} \textbf{true}::\textbf{true} \rightarrow [[\mathcal{C},\ B_{p+1}\ ]]$$

Simple pattern matching:

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l}\textbf{case}\ E'\ \textbf{in}\\ \quad P_1^0,\ldots,P_{n_0}^0\\ \qquad \textbf{when}\ E_0 \rightarrow B_0'\\ \quad \ldots\\ \quad P_1^p,\ldots,P_{n_p}^p\\ \qquad \textbf{when}\ E_p \rightarrow B_p'\\ \quad \textbf{otherwise}\ B_{p+1}'\\ \textbf{endcase}\end{array}\right)\right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{case}\ [[\mathcal{C},\ E'\ ]]::(\&X)\rightarrow\\ \quad \textbf{case}\\ \qquad X::[[\mathcal{C},\ P_1^0\ ]]\ \textbf{or}\ \ldots\ \textbf{or}\ X::[[\mathcal{C},\ P_{n_0}^0\ ]]\\ \qquad\quad \textbf{when}\ [[\mathcal{C},\ E_0\ ]] \rightarrow [[\mathcal{C},\ B_0'\ ]]\\ \qquad \ldots\\ \qquad X::[[\mathcal{C},\ P_1^p\ ]]\ \textbf{or}\ \ldots\ \textbf{or}\ X::[[\mathcal{C},\ P_{n_p}^p\ ]]\\ \qquad\quad \textbf{when}\ [[\mathcal{C},\ E_p\ ]] \rightarrow [[\mathcal{C},\ B_p'\ ]]\\ \qquad \textbf{true}::\textbf{true} \rightarrow [[\mathcal{C},\ B_{p+1}'\ ]]\\ \quad \textbf{endcase}\\ \textbf{endcase}\end{array}\right)$$

Conditional behaviour:

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l}\textbf{if}\ E_0\ \textbf{then}\ B_0'\\ \textbf{elsif}\ E_1\ \textbf{then}\ B_1'\\ \ldots\\ \textbf{elsif}\ E_n\ \textbf{then}\ B_n'\\ \textbf{else}\ B_{n+1}'\\ \textbf{endif}\end{array}\right)\right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l}\textbf{case}\\ \quad [[\mathcal{C},\ E_0\ ]]::\textbf{true} \rightarrow [[\mathcal{C},\ B_0'\ ]]\\ \quad [[\mathcal{C},\ E_1\ ]]::\textbf{true} \rightarrow [[\mathcal{C},\ B_1'\ ]]\\ \quad \ldots\\ \quad [[\mathcal{C},\ E_n\ ]]::\textbf{true} \rightarrow [[\mathcal{C},\ B_n'\ ]]\\ \quad \textbf{true}::\textbf{true} \rightarrow B_{n+1}'\\ \textbf{endcase}\end{array}\right)$$

Guarded behaviour:

$$[[\mathcal{C},\ [E_{\mathrm{bool}}]\ \texttt{->}\ B_0\ ]] \stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{case} \\ \quad [[\mathcal{C},\ E_{\mathrm{bool}}\ ]]\texttt{::}\textbf{true} \rightarrow [[\mathcal{C},\ B_0\ ]] \\ \quad \textbf{true}\texttt{::}\textbf{true} \rightarrow \textbf{stop} \\ \textbf{endcase} \end{array} \right)$$

Old let behaviour:

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l} \textbf{let}\ V_1\texttt{:}T_1\texttt{:=}E_1,\ldots, \\ \qquad V_n\texttt{:}T_n\texttt{:=}E_n \\ \textbf{in}\ B \end{array} \right)\right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{local}\ V_1\texttt{:}T_1,\ldots,V_n\texttt{:}T_n\ \textbf{in} \\ \quad \textbf{case} \\ \quad\quad [[\mathcal{C},\ E_1\ ]]\texttt{::}(\&V_1)\ \textbf{and}\ \ldots\ \textbf{and} \\ \quad\quad [[\mathcal{C},\ E_n\ ]]\texttt{::}(\&V_n) \rightarrow [[\mathcal{C},\ B\ ]] \\ \quad \textbf{endcase} \\ \textbf{endloc} \end{array} \right)$$

Hiding:

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l} \textbf{hide}\ G_1\texttt{:}\Gamma_1,\ldots,G_n\texttt{:}\Gamma \\ \textbf{in}\ B \\ \textbf{endhide} \end{array} \right)\right]\right] \stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{hide}\ G_1\texttt{:}\Gamma_1,\ldots,G_n\texttt{:}\Gamma \\ \textbf{in}\ [[\mathcal{C}',\ B\ ]] \\ \textbf{endhide} \end{array} \right)$$

where $\mathcal{C}' = \mathcal{C} + (G_1 \mapsto \mathcal{C}(\Gamma_1)) + \ldots + (G_n \mapsto \mathcal{C}(\Gamma_n))$.

Disabling:

$$[[\mathcal{C},\ \textbf{dis}\ B_1\,\texttt{[>}B_2\ \textbf{enddis}\,]] \stackrel{\texttt{def}}{=} [[\mathcal{C},\ B_1\ ]]\texttt{[>}[[\mathcal{C},\ B_2\ ]]$$

Process instantiation: similarly to function instantiation translation.

Post-renaming:

$$\left[\left[\mathcal{C},\ \left(\begin{array}{l} \textbf{rename} \\ \quad G_0[(V_1^0\texttt{:}T_1^0,\ldots V_{n_0}^0\texttt{:}T_{n_0}^0)]\ \ \texttt{:=}\ \ G_0'[(V_1'^0\texttt{:=}E_1'^0,\ldots,V_{m_0}'^0\texttt{:=}E_{m_0}'^0)] \\ \quad \ldots \\ \quad G_p[(V_1^p\texttt{:}T_1^p,\ldots V_{n_p}^p\texttt{:}T_{n_p}^p)]\ \ \texttt{:=}\ \ G_p'[(V_1'^p\texttt{:=}E_1'^p,\ldots,V_{m_p}'^p\texttt{:=}E_{m_p}'^p)] \\ \textbf{in}\ B \\ \textbf{endren} \end{array} \right)\right]\right]$$

$$\stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{rename} \\ \quad G_0[(V_1^0\texttt{:}T_1^0,\ldots V_{n_0}^0\texttt{:}T_{n_0}^0)]\ \ \texttt{:=}\ \ [[\mathcal{C},\ G_0'[(V_1'^0\texttt{:=}E_1'^0,\ldots,V_{m_0}'^0\texttt{:=}E_{m_0}'^0)]\ ]] \\ \quad \ldots \\ \quad G_p[(V_1^p\texttt{:}T_1^p,\ldots V_{n_p}^p\texttt{:}T_{n_p}^p)]\ \ \texttt{:=}\ \ [[\mathcal{C},\ G_p'[(V_1'^p\texttt{:=}E_1'^p,\ldots,V_{m_p}'^p\texttt{:=}E_{m_p}'^p)]\ ]] \\ \textbf{in}\ [[\mathcal{C}',\ B\ ]] \\ \textbf{endren} \end{array} \right)$$

where

$$\mathcal{C}' = \mathcal{C} + (G_0 \mapsto cht_0) + (G_p \mapsto cht_p)$$

$$\forall i \in 0..p \quad cht_i = \left\{ \begin{array}{ll} \{(V_1^i : T_1^i : in, \ldots V_{m_i}^i : T_{m_i}^i : in)\} & \textbf{iff}\ G_i\ \textit{has arguments} \\ \mathcal{C}(G_i') & \textbf{otherwise} \end{array} \right.$$

4.7 Static semantics of the reduced user language

4.7.1 Notations

When A and B are sets, Fin(A) denotes the set of finite subsets of A, and $A \stackrel{\text{fin}}{\rightarrow} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B. The domain and range of a finite map, f, are denoted Dom(f) and Ran(f). A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$; in particular, the empty map is $\{\}$.

When f and g are finite maps we define:

- $f + g$ with domain $\text{Dom}(f) \cup \text{Dom}(g)$ and values:

$$(f+g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \text{Dom}(g) \\ g(a) & \textbf{if } a \notin \text{Dom}(f) \\ error & \textbf{otherwise} \end{cases}$$

  This operator is called *disjunct* composition of f and g.

- $f \oplus g$ with domain $\text{Dom}(f) \cup \text{Dom}(g)$ and values:

$$(f \oplus g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \text{Dom}(g) \\ g(a) & \textbf{otherwise} \end{cases}$$

  This operator is called f *modified by* (or *overridden*) g.

- $f \odot g$ with domain $\text{Dom}(f) \cup \text{Dom}(g)$ and values:

$$(f \odot g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \text{Dom}(g) \\ g(a) & \textbf{if } a \notin \text{Dom}(f) \\ f(a) & \textbf{if } g(a) = f(a) \\ error & \textbf{otherwise} \end{cases}$$

  This operator is therefore called *match* composition of f and g.

- $f \ominus \{a_1, ..., a_n\}$ where $\{a_1, ..., a_n\} \subset \text{Dom}(f)$, with domain $\text{Dom}(f) \setminus \{a_1, ..., a_n\}$ and values:

$$(f \ominus \{a_1, ..., a_n\})(a) = f(a) \textbf{ if } a \notin \{a_1, ..., a_n\}$$

  This operator is usually called restriction of domain of f.

When the range of a partial map is a finite set of subsets, $A \stackrel{\text{sfin}}{\rightarrow} \text{Fin}(B)$ denotes *finite set-maps* from A to B. When A is a set, List (A) denotes the set of finite lists with elements in A. A finite list will often be written explicitly in the form $(a_1, \cdots, a_k)$, $k \geq 0$. The empty list is (), and list concatenation is noted &.

4.7.2 Semantical objects

In addition to the class of *syntactic* objects, defined in sub-section 4.2, there are classes of *semantic* objects used to describe the meaning of syntactic objects.

All semantics objects in the static semantics are built from identifiers. With each special constant K we associate a type name *type*(K) which is either **integer**, **bool**, **real** (iff we suppose that this will be the built-in types). The **void** constant is used to treat functions and procedures with "**out**" parameters.

The classes of semantic objects are:

$$
\begin{array}{rcll}
T & \in & \mathrm{TyName} = \mathrm{Typ} \cup \{\mathbf{integer}, \mathbf{bool}, \mathbf{real}\} & \textit{type names} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyName} \xrightarrow{\mathrm{fin}} \mathrm{TyName} & \textit{type environment} \\
tl & \in & \mathrm{TuplType} = \mathrm{List}(\mathrm{TyName}) & \textit{tuple of type} \\
VE & \in & \mathrm{VarEnv} = \mathrm{Var} \xrightarrow{\mathrm{fin}} (\mathrm{TyName} \times \{def, undef\}) & \textit{variable environment} \\
vl & \in & \mathrm{VarList} = \mathrm{List}(\mathrm{Var} \times \mathrm{Typ} \times \{in, out\}) & \textit{typed variable list} \\
ct & \in & \mathrm{ConType} = \mathrm{VarList} \times \mathrm{TyName} \times \mathrm{UIDs} & \textit{type of constructors} \\
CE & \in & \mathrm{ConEnv} = \mathrm{Con} \xrightarrow{\mathrm{fin}} \mathrm{Fin}(\mathrm{ConType}) & \textit{environment of constructors} \\
cht & \in & \mathrm{ChType} = \mathrm{Fin}(\mathrm{VarList}) & \textit{union of labeled tuples} \\
ChE & \in & \mathrm{ChEnv} = \mathrm{ChName} \xrightarrow{\mathrm{fin}} \mathrm{ChType} & \textit{channel environment} \\
GE & \in & \mathrm{GatEnv} = \mathrm{Gat} \xrightarrow{\mathrm{fin}} \mathrm{ChType} & \textit{gate environment} \\
gl & \in & \mathrm{GatList} = \mathrm{List}(\mathrm{Gat} \times \mathrm{ChType}) & \textit{typed gate list} \\
ft & \in & \mathrm{FunType} = \mathrm{GatList} \times \mathrm{VarList} \times \mathrm{TuplType} \times \mathrm{UIDs} & \textit{type of functions} \\
FE & \in & \mathrm{FunEnv} = \mathrm{Fun} \xrightarrow{\mathrm{fin}} \mathrm{Fin}(\mathrm{FunType}) & \textit{enviroment of functions} \\
pt & \in & \mathrm{ProcType} = \mathrm{GatList} \times \mathrm{VarList} \times \mathrm{TuplType} & \textit{type of processes} \\
PE & \in & \mathrm{ProcEnv} = \mathrm{Proc} \xrightarrow{\mathrm{fin}} \mathrm{ProcType} & \textit{enviroment of processes} \\
\mathcal{C} & \in & \mathrm{Context} = \mathrm{TyEnv} \times \mathrm{VarEnv} \times \mathrm{ConEnv} \times \mathrm{ChEnv} \times & \textit{context} \\
 & & \qquad \mathrm{GatEnv} \times \mathrm{FunEnv} \times \mathrm{ProcEnv} &
\end{array}
$$

We give below the notations used for objects defined above:

$$
\begin{array}{rcll}
A & ::= & in & \\
 & | & out & \\
tl & ::= & (T_1, \ldots, T_k) & \\
vl & ::= & (V_1 : T_1 : A_1, \ldots, V_k : T_k : A_k) & \\
cht & ::= & \{(V_1^1 : T_1^1 : A_1, \ldots, V_{k_1}^1 : T_{k_1}^1 : A_{k_1}^1), \ldots, (V_1^n : T_1^n : A_1, \ldots, V_{k_n}^n : T_{k_n}^n : A_{k_n}^n)\} & \\
gl & ::= & (G_1 : cht, \ldots, G_k : cht) & \\
profile & ::= & vl \rightarrow (T) \rightarrow uid & \textit{profile of constructors} \\
 & | & gl \rightarrow vl \rightarrow tl \rightarrow uid & \textit{function profile} \\
d & ::= & def & \\
 & | & undef & \\
\mathcal{C} & ::= & T \mapsto T' & \\
 & | & V \mapsto (T, d) & \\
 & | & C \mapsto \{profile_1, \ldots, profile_n\} & \\
 & | & \Gamma \mapsto cht & \\
 & | & G \mapsto cht & \\
 & | & F \mapsto \{profile_1, \ldots, profile_n\} & \\
 & | & \Pi \mapsto gl \rightarrow vl \rightarrow tl &
\end{array}
$$

| $\{\}$
| $\mathcal{C} + \mathcal{C}$

We write $\mathcal{C} \vdash T \Rightarrow T'$ for $\mathcal{C} = \mathcal{C}' + (T \mapsto T', \{\}, \{\}, \{\}, \{\}, \{\}, \{\})$, and similarly for other bindings.

For contexts of type, variable, channel, gates, and process bindings, "+" is disjoint composition of finite maps. For contexts of constructor bindings and function bindings, the operation "+" is the disjoint composition of set finite maps as defined in section 4.6.1.

Remark: The context presented in this section is a superset of context presented in section 4.6.1, because it includes variable binding.

The operator $\oplus$ is used for context overriding, that is:

$$\mathcal{C} \oplus \{\} = \mathcal{C} \qquad (\mathcal{C}[+T \mapsto T_1]) \oplus (\mathcal{C}' + T \mapsto T_2) = (\mathcal{C} + T \mapsto T_2) \oplus \mathcal{C}'$$

and similarly for the other bindings.

In the presentation of the rules, phrases within single brackets $\langle\rangle$ are called *first options*. To reduce the number of rules, we have adopted the following convention:

In each instance of a rule, the options must be either all present or all absent.

4.7.3 Static semantics of variable lists

To simplify the presentation we introduce the syntactic domain VList, having values:

VL ::=		*empty list*
|	$(V_1 : T'_1, ..., V_n : T'_n)$	*non-attributed list*
|	$(\mathbf{in} \mid \mathbf{out} V_1 : T_1, ..., \mathbf{in} \mid \mathbf{out}\ V_n : T_n)$	*attributed list*

These lists appears in the declaration of types, constructors, channels, gates, functions and processes.

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash VL \Rightarrow vl, \mathcal{C}_{\text{in}}, \mathcal{C}_{\text{out}}$$

meaning "In the context $\mathcal{C}$, the variable list VL is well formed and gives the list vl and the context of defined variable $\mathcal{C}_{\text{in}}$, and the context of undefined variables $\mathcal{C}_{\text{out}}$"

Static semantics:

$$\frac{}{\mathcal{C} \vdash () \Rightarrow ()} \tag{1}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow T'_1 \quad \cdots \quad \mathcal{C} \vdash T_n \Rightarrow T'_n \\ (+_{i=1}^{i=n} V_i \mapsto (T'_i, def)) \neq error\end{array}}{\begin{array}{c}\mathcal{C} \vdash (V_1 : T'_1, ..., V_n : T'_n) \Rightarrow \\ (V_1 : T'_1 : in, \ldots, V_n : T'_n : in),\ (+_{i=1}^{i=n} V_i \mapsto (T'_i, def)), \{\}\end{array}} \tag{2}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow T_1' \quad \cdots \quad \mathcal{C} \vdash T_n \Rightarrow T_n' \\ (+_{i=1}^{i=n} V_i \mapsto (T_i', d_i)) \neq \mathit{error}\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\mathbf{in} \mid \mathbf{out} V_1 : T_1, ..., \mathbf{in} \mid \mathbf{out}\ V_n : T_n) \Rightarrow \\ (V_1 : T_1' : A_1, \ldots, V_n : T_n' : A_n), \mathcal{C}_{\text{in}}, \mathcal{C}_{\text{out}}\end{array}} \tag{3}$$

where

$$A_i = \begin{cases} \mathit{out} & \textbf{if out } V_i : T_i \\ \mathit{in} & \textbf{otherwise} \end{cases}$$
$$d_i = \begin{cases} \mathit{undef} & \textbf{if out } V_i : T_i \\ \mathit{def} & \textbf{otherwise} \end{cases}$$
$$\mathcal{C}_{\text{in}} = \{V_i \mapsto (T_i', \mathit{def}) \mid \mathbf{in}\ V_i : T_i\}$$
$$\mathcal{C}_{\text{out}} = \{V_i \mapsto (T_i', \mathit{undef}) \mid \mathbf{out}\ V_i : T_i\}$$

4.7.4 Static semantics of gate lists

To simplify the presentation we introduce the syntactic domain GList, having values:

GL ::=		*empty list*
	$\mid \quad G_1 : \Gamma_1, ..., G_n : \Gamma_n$	*non-empty list*

These lists appears in the declaration of functions and processes.
The static semantics is given by assertions of form:

$$\mathcal{C} \vdash GL \Rightarrow gl$$

meaning "In the context $\mathcal{C}$, the gate list GL is well formed and gives the list gl"
Static semantics:

$$\frac{}{\mathcal{C} \vdash () \Rightarrow ()} \tag{4}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \Gamma_1 \Rightarrow \mathit{cht}_1 \quad \cdots \quad \mathcal{C} \vdash \Gamma_n \Rightarrow \mathit{cht}_n \\ (+_{i=1}^{i=n} G_i \mapsto \mathit{cht}_i) \neq \mathit{error}\end{array}}{\begin{array}{c}\mathcal{C} \vdash (G_1 : \Gamma_1, ..., G_n : \Gamma_n) \Rightarrow \\ (G_1 : \mathit{cht}_1, \ldots, G_n : \mathit{cht}_n),\ (+_{i=1}^{i=n} G_i \mapsto \mathit{cht}_i)\end{array}} \tag{5}$$

4.7.5 Static semantics of declarations

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash D \Rightarrow \mathcal{C}'$$

meaning "In the context $\mathcal{C}$, the declaration D is well formed and gives context $\mathcal{C}'$."

$$\frac{\mathcal{C} \vdash T' \Rightarrow T''}{\mathcal{C} \vdash (\textbf{type } T \textbf{ is } T' \textbf{ endtype}) \Rightarrow (T \Rightarrow T'')} \tag{6}$$

$$\frac{\begin{array}{c}\mathcal{C} + (T \mapsto T) \vdash VL_1 \Rightarrow vl_1, \mathcal{C}_1, \{\} \quad \ldots \quad \mathcal{C} + (T \mapsto T) \vdash VL_p \Rightarrow vl_p, \mathcal{C}_p, \{\} \\ (\mathcal{C}_1 \odot \ldots \odot \mathcal{C}_p) \neq \mathit{error}\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\textbf{type } T \textbf{ is } C_1 VL_1 \ldots C_p \, VL_p \textbf{ endtype}) \Rightarrow \\ (T \mapsto T, C_1 \mapsto vl_1 \rightarrow (T) \rightarrow \mathit{newuid}_1, \ldots, C_p \mapsto vl_p \rightarrow (T) \rightarrow \mathit{newuid}_p)\end{array}} \tag{7}$$

$$\frac{\mathcal{C} \vdash VL \Rightarrow vl, \mathcal{C}', \{\}}{\mathcal{C} \vdash (\textbf{exception } \Xi \textbf{ is } VL \textbf{ endexc}) \Rightarrow (\Xi \Rightarrow \{vl\})} \tag{8}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash VL_1 \Rightarrow vl_1, \mathcal{C}_1, \{\} \quad \ldots \quad \ldots \quad \mathcal{C} \vdash VL_p \Rightarrow vl_p, \mathcal{C}_p, \{\} \\ (\mathcal{C}_1 \odot \ldots \odot \mathcal{C}_p) \neq \mathit{error}\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\textbf{channel } \Gamma \textbf{ is } VL_1 \ldots VL_p \textbf{ endchan}) \Rightarrow \\ (\Gamma \Rightarrow \{vl_1, \ldots, vl_p\})\end{array}} \tag{9}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash GL \Rightarrow gl, \mathcal{C}_X \\ \mathcal{C} \vdash VL \Rightarrow vl, \mathcal{C}_{\text{in}}, \mathcal{C}_{\text{out}} \\ \langle \mathcal{C} \vdash T \Rightarrow T' \rangle \\ (\mathcal{C} \oplus \mathcal{C}_X \oplus \mathcal{C}_{\text{in}} \oplus \mathcal{C}_{\text{out}}) + \\ +(F \mapsto gl \rightarrow vl \rightarrow (\ \langle T' \rangle\ ) \rightarrow \mathit{newuid}) \vdash E \Rightarrow \mathcal{C}', (\ \langle T' \rangle\ ) \\ \text{Dom}(\mathcal{C}') = \text{Dom}(\mathcal{C}_{\text{out}})\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\textbf{function } F\texttt{[}GL\texttt{]}VL \ \langle :T \rangle \ \textbf{ is } E \textbf{ endfunc}) \Rightarrow \\ (F \mapsto gl \rightarrow vl \rightarrow (\ \langle T' \rangle\ ) \rightarrow \mathit{newuid})\end{array}} \tag{10}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash GL \Rightarrow gl, \mathcal{C}_X \qquad \mathcal{C} \vdash VL \Rightarrow vl, \mathcal{C}_{\text{in}}, \mathcal{C}_{\text{out}} \\ \langle \mathcal{C} \vdash T_1 \Rightarrow T_1' \quad \cdots \quad \mathcal{C} \vdash T_n \Rightarrow T_n' \rangle \\ (\mathcal{C} \oplus \mathcal{C}_G \oplus \mathcal{C}_{\text{in}} \oplus \mathcal{C}_{\text{out}}) + \\ +(\Pi \mapsto gl \rightarrow vl \rightarrow (\ \langle T_1', \ldots, T_m' \rangle\ ) \vdash B \Rightarrow \mathcal{C}', (\ \langle T_1', \ldots, T_m' \rangle\ ) \\ \text{Dom}(\mathcal{C}') = \text{Dom}(\mathcal{C}_{\text{out}})\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\textbf{process } \Pi\texttt{[}GL\texttt{]}VL \ \langle :\ \textbf{exit}\ (T_1, \ldots, T_m) \rangle \ \textbf{ is } B \textbf{ endproc}) \Rightarrow \\ (P \mapsto gl \rightarrow vl \rightarrow (\ \langle T_1', \ldots, T_m' \rangle\ ))\end{array}} \tag{11}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash GL \Rightarrow gl, \mathcal{C}_X \qquad \mathcal{C} \vdash VL \Rightarrow vl, \mathcal{C}_{\text{in}}, \{\} \\ \mathcal{C} \oplus \mathcal{C}_G \oplus \mathcal{C}_{\text{in}} + (P \mapsto gl \rightarrow vl \rightarrow ()) \vdash B \Rightarrow \{\}, ()\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\textbf{process } \Pi\texttt{[}GL\texttt{]}VL\textbf{:noexit is } \ \textbf{endproc}) \Rightarrow \\ (P \mapsto gl \rightarrow vl \rightarrow ())\end{array}} \tag{12}$$

$$\frac{\mathcal{C} \vdash D_1 \Rightarrow \mathcal{C}_1 \quad \mathcal{C} \Rightarrow \mathcal{C}_2}{\mathcal{C} \vdash D_1 D_2 \Rightarrow \mathcal{C}_1 + \mathcal{C}_2} \tag{13}$$

$$\frac{}{\mathcal{C} \vdash \Rightarrow \{\}} \tag{14}$$

4.7.6 Static semantics of patterns

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow \mathcal{C}'$$

meaning "In the context $\mathcal{C}$, matching pattern P to type T gives the context $\mathcal{C}'$."

$$\frac{}{\mathcal{C} \vdash (K \Rightarrow T) \Rightarrow \{\}}[K : T] \tag{15}$$

$$\frac{\mathcal{C} \vdash V \Rightarrow (T, undef)}{\mathcal{C} \vdash (V \Rightarrow T) \Rightarrow (V \mapsto (T, def))} \tag{16}$$

$$\frac{}{\mathcal{C} \vdash (\mathbf{any}\ T \Rightarrow T) \Rightarrow \{\}} \tag{17}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash C \Rightarrow (V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n) \rightarrow T \\ \mathcal{C} \vdash (P_1 \Rightarrow T_1) \Rightarrow \mathcal{C}_1 \quad \ldots \mathcal{C} \vdash (P_n \Rightarrow T_n) \Rightarrow \mathcal{C}_n\end{array}}{\mathcal{C} \vdash (C(P_1, \ldots, P_n) \Rightarrow T) \Rightarrow \mathcal{C}_1 + \ldots + \mathcal{C}_n} \tag{18}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T \Rightarrow T' \\ \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow \mathcal{C}'\end{array}}{\mathcal{C} \vdash (P\ \mathbf{of}\ T \Rightarrow T') \Rightarrow \mathcal{C}'} \tag{19}$$

4.7.7 Static semantics of match expressions

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash \mathcal{C} \vdash M \Rightarrow \mathcal{C}'$$

meaning "In the context $\mathcal{C}$, the match expression M gives the context $\mathcal{C}'$."

$$\frac{\begin{array}{c}\mathcal{C} \vdash E \Rightarrow \{\}, T \\ \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow \mathcal{C}'\end{array}}{\mathcal{C} \vdash (E :: P) \Rightarrow \mathcal{C}'} \tag{20}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash M_1 \Rightarrow \mathcal{C}' \\ \mathcal{C} + \mathcal{C}' \vdash E \Rightarrow \{\}, \mathbf{bool}\end{array}}{\mathcal{C} \vdash (M_1 \ \mathbf{when} \ E) \Rightarrow \mathcal{C}'} \tag{21}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash M_1 \Rightarrow \mathcal{C}_1 \\ \mathcal{C} \vdash M_2 \Rightarrow \mathcal{C}_2\end{array}}{\mathcal{C} \vdash (M_1 \ \mathbf{and} \ M_2) \Rightarrow \mathcal{C}_1 + \mathcal{C}_2} \tag{22}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash M_1 \Rightarrow \mathcal{C}' \\ \mathcal{C} \vdash M_2 \Rightarrow \mathcal{C}'\end{array}}{\mathcal{C} \vdash (M_1 \ \mathbf{or} \ M_2) \Rightarrow \mathcal{C}'} \tag{23}$$

4.7.8 Static semantics of value expressions

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash E \Rightarrow \mathcal{C}', tl$$

meaning "In the context $\mathcal{C}$, the expression E gives the context $\mathcal{C}'$ and has the type tl."

$$\frac{}{\mathcal{C} \vdash K \Rightarrow \{\}, (T)}[K : T] \tag{24}$$

$$\frac{\mathcal{C} \vdash V \Rightarrow (T, def)}{\mathcal{C} \vdash V \Rightarrow \{\}, (T)} \tag{25}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash C \Rightarrow (V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n) \rightarrow T \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, (T_1) \quad \ldots \mathcal{C} \vdash E_n \Rightarrow \{\}, (T_n)\end{array}}{\mathcal{C} \vdash (C(E_1, \ldots, E_n)) \Rightarrow \{\}, (T)} \tag{26}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash F \Rightarrow (X_1 : cht_1, \ldots, X_p : cht_p) \rightarrow (V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n) \rightarrow tl \rightarrow uid \\ \mathcal{C} \vdash X'_1 \Rightarrow cht_1 \quad \ldots \quad \mathcal{C} \vdash X'_p \Rightarrow cht_p \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, T_1 \quad | \quad \mathcal{C} \vdash (P_1 \Rightarrow T_1) \Rightarrow \mathcal{C}_1 \quad \ldots \\ \mathcal{C} \vdash E_n \Rightarrow \{\}, T_n \quad | \quad \mathcal{C} \vdash (P_n \Rightarrow T_n) \Rightarrow \mathcal{C}_n\end{array}}{\begin{array}{c}\mathcal{C} \vdash (F\,\texttt{[}X'_1, \ldots, X'_p\texttt{]}\,(E_1 \mid \texttt{\&}P_1, ..., E_n \mid \texttt{\&}P_n)) \Rightarrow \\ (\{\} \mid \mathcal{C}_1) + \ldots + (\{\} \mid \mathcal{C}_n), tl\end{array}} \tag{27}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash M_0 \Rightarrow \mathcal{C}_0 \quad \mathcal{C} \oplus \mathcal{C}_0 \vdash E_0 \Rightarrow \mathcal{C}', tl \quad \ldots \\ \mathcal{C} \vdash M_p \Rightarrow \mathcal{C}_p \quad \mathcal{C} \oplus \mathcal{C}_p \vdash E_p \Rightarrow \mathcal{C}', tl\end{array}}{\mathcal{C} \vdash (\mathbf{case} \ M_0 \rightarrow E_0 \ldots M_p \rightarrow E_p \ \mathbf{endcase}) \Rightarrow \mathcal{C}', tl} \tag{28}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash X \Rightarrow \{(\ \langle V_1 : T_1 : A_1, \ldots, V_p : T_p : A_p \rangle\ )\} \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, (\ \langle T_1 \rangle\ ) \quad \ldots \quad \mathcal{C} \vdash E_p \Rightarrow \{\}, (\ \langle T_p \rangle\ )\end{array}}{\mathcal{C} \vdash (\mathbf{raise}\ X\ \langle (E_1, \ldots, E_p) \rangle\ ) \Rightarrow \{\}, ()} \tag{29}$$

$$\frac{\begin{array}{c}\langle \mathcal{C} \vdash VL_1 \Rightarrow vl_1, \mathcal{C}_1, \{\} \quad \ldots \quad \mathcal{C} \vdash VL_n \Rightarrow vl_n, \mathcal{C}_n, \{\}\rangle \\ \mathcal{C} \oplus (+_{i=1}^{i=p}(X_i \mapsto \{\ \langle vl_i \rangle\ \})) \vdash E \Rightarrow \mathcal{C}', tl \\ \mathcal{C}\ \langle \oplus \mathcal{C}_1 \rangle\ \vdash E_1 \Rightarrow \mathcal{C}', tl' \quad \ldots \quad \mathcal{C}\ \langle \oplus \mathcal{C}_n \rangle\ \vdash E_n \Rightarrow \mathcal{C}', tl'\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\mathbf{trap}\ X_1\ \langle VL_1 \rangle\ \rightarrow E_1\ \ldots\ X_n\ \langle VL_n \rangle\ \rightarrow E_n\ \mathbf{in}\ E\ \mathbf{endtrap}) \\ \Rightarrow \mathcal{C}', tl\end{array}} [\delta \notin \{X_1, .., X_n\}] \tag{30}$$

$$\frac{\begin{array}{c}\langle \mathcal{C} \vdash VL_1 \Rightarrow vl_1, \mathcal{C}_1, \{\} \quad \ldots \mathcal{C} \vdash VL_n \Rightarrow vl_n, \mathcal{C}_n, \{\}\rangle \\ \mathcal{C} \oplus (+_{i=1}^{i=p}(X_i \mapsto \{\ \langle vl_i \rangle\ \})) \vdash E \Rightarrow \mathcal{C}', tl' \\ \mathcal{C}' = \{\}\ \langle \mathcal{C}_1 \rangle \\ \mathcal{C} \oplus \mathcal{C}_1 \vdash E_1 \Rightarrow \mathcal{C}'', tl'' \quad \ldots \quad \mathcal{C} \oplus \mathcal{C}_n \vdash E_n \Rightarrow \mathcal{C}'', tl''\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\mathbf{trap}\ X_1\ \langle VL_1 \rangle\ \rightarrow E_1\ \ldots\ X_n\ \langle VL_n \rangle\ \rightarrow E_n\ \mathbf{in}\ E\ \mathbf{endtrap}) \\ \Rightarrow \mathcal{C}'', tl''\end{array}} [\delta = X_1] \tag{31}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow T_1' \quad \ldots \quad \mathcal{C} \vdash T_n \Rightarrow T_n' \\ \mathcal{C} \oplus (+_{i=1}^{i=n} V_i \mapsto (T_i', undef)) \vdash E \Rightarrow \mathcal{C}', tl\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\mathbf{local}\ V_1 : T_1, \ldots, V_n : T_n\ \mathbf{in}\ E\ \mathbf{endloc}) \Rightarrow \\ \mathcal{C}' \ominus \{V_1, \ldots, V_n\}, tl\end{array}} \tag{32}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash E \Rightarrow \{\}, (T) \\ \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow \mathcal{C}'\end{array}}{\mathcal{C} \vdash (P := E) \Rightarrow \mathcal{C}', ()} \tag{33}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash E_1 \Rightarrow \mathcal{C}_1, () \\ \mathcal{C} \oplus \mathcal{C}_1 \Rightarrow \mathcal{C}_2, tl\end{array}}{\mathcal{C} \vdash (E_1\ \ ;\ \ E_2) \Rightarrow \mathcal{C}_1 \oplus \mathcal{C}_2, tl} \tag{34}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash E_1 \Rightarrow \{\}, (T) \\ \mathcal{C} \vdash E_2 \Rightarrow \{\}, (T)\end{array}}{\mathcal{C} \vdash (E_1\ \ =\ \ E_2) \Rightarrow \{\}, (T)} \tag{35}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash E_0 \Rightarrow \{\}, (T) \\ \mathcal{C} \vdash _ \Rightarrow \{((\ldots V_0 : T_0 : in \ldots) \rightarrow (T) \rightarrow _)\}\end{array}}{\mathcal{C} \vdash (E_0\,.\,V_0) \Rightarrow \{\}, (T_0)} \tag{36}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash E_0 \Rightarrow \{\}, (T) \\ \mathcal{C} \vdash _ \Rightarrow (V'_1 : T'_1 : A_1, \ldots, V'_m : T'_m : A_m) \rightarrow T \rightarrow _ \\ (m \geq n) \ \wedge \ (\forall i \in 1..n \quad (V'_{\rho(i)} = V_i) \\ \mathcal{C} \oplus (+_{i=1}^{i=n} V'_i \mapsto (T'_i, def)) \vdash E_1 \Rightarrow \{\}, (T'_1) \quad \ldots \\ \mathcal{C} \oplus (+_{i=1}^{i=n} V'_i \mapsto (T'_i, def)) \vdash E_n \Rightarrow \{\}, (T'_n) \end{array}}{\mathcal{C} \vdash (E_0 \,.\, \{V_1 \,\texttt{:=}\, E_1, \ldots, V_n \,\texttt{:=}\, E_n\}) \Rightarrow \{\}, (T)} [\rho \in P(m)] \tag{37}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash T_1 \Rightarrow T'_1 \quad \ldots \quad \mathcal{C} \vdash T_n \Rightarrow T'_n \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, T'_1 \quad \ldots \quad \mathcal{C} \vdash E_n \Rightarrow \{\}, T'_n \\ \mathcal{C} \oplus (+_{j=1}^{j=n} V_j \mapsto (T'_j, def)) \vdash E_0 \Rightarrow \mathcal{C}', T' \end{array}}{\mathcal{C} \vdash \left(\begin{array}{l} \textbf{loop } V_1 \,\texttt{:}\, T_1 \,\texttt{:=}\, E_1, \ldots, V_n \,\texttt{:}\, T_n \,\texttt{:=}\, E_n \\ \textbf{in } E_0 \\ \textbf{endloop} \end{array} \right) \Rightarrow \mathcal{C}', T'} \tag{38}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash V_1 \Rightarrow (T_1, def) \quad \ldots \quad \mathcal{C} \vdash V_n \Rightarrow (T_n, def) \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, T_1 \quad \ldots \quad \mathcal{C} \vdash E_n \Rightarrow \{\}, T_n \end{array}}{\mathcal{C} \vdash (\textbf{continue} V_1 \,\texttt{:=}\, E_1, \ldots, V_n \,\texttt{:=}\, E_n) \Rightarrow \{\}, ()} \tag{39}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash T \Rightarrow T' \\ \mathcal{C} \vdash E_0 \Rightarrow \{\}, (T') \end{array}}{\mathcal{C} \vdash (E_0 \ \textbf{of} \ T) \Rightarrow \{\}, (T')} \tag{40}$$

4.7.9 Static semantics of behaviour expressions

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash \mathcal{C} \vdash B \Rightarrow \mathcal{C}', tl$$

meaning "In the context $\mathcal{C}$, the behaviour expression B gives the context $\mathcal{C}'$ and has the type list tl."

$$\frac{\mathcal{C} \vdash E_1 \Rightarrow \{\}, (T_1) \quad \ldots \quad \mathcal{C} \vdash E_n \Rightarrow \{\}, (T_n)}{\mathcal{C} \vdash (\delta(E_1, \ldots, E_n)) \Rightarrow \{\}, (T_1, \ldots, T_n)} \tag{41}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash G \Rightarrow (V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n) \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, (T_1) \quad | \quad \mathcal{C} \vdash (P_1 \Rightarrow T_1) \Rightarrow \mathcal{C}_1 \quad \ldots \\ \mathcal{C} \vdash E_n \Rightarrow \{\}, (T_n) \quad | \quad \mathcal{C} \vdash (P_n \Rightarrow T_n) \Rightarrow \mathcal{C}_n \\ +_{i=1}^{i=n} (\{\} \mid \mathcal{C}_i) \neq error \\ \langle \mathcal{C} \oplus (+_{i=1}^{i=n} (\{\} \mid \mathcal{C}_i) \vdash E \Rightarrow \{\}, (\textbf{bool}) \rangle \end{array}}{\begin{array}{c} \mathcal{C} \vdash (G(E_1 \mid \texttt{\&} P_1, \ldots, E_n \mid \texttt{\&} P_n) \ \langle \texttt{[} E \texttt{]} \rangle \ ) \\ \Rightarrow (+_{i=1}^{i=n} (\{\} \mid \mathcal{C}_i)), () \end{array}} \tag{42}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash E \Rightarrow \{\}, (T) \\ \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow \mathcal{C}' \end{array}}{\mathcal{C} \vdash (P \,\texttt{:=}\, E) \Rightarrow \mathcal{C}', ()} \tag{43}$$

$$\frac{\begin{array}{c}\langle \mathcal{C} \vdash VL_1 \Rightarrow vl_1, \mathcal{C}_1, \{\} \quad \ldots \quad \mathcal{C} \vdash VL_n \Rightarrow vl_n, \mathcal{C}_n, \{\}\rangle \\ \mathcal{C} \oplus (+_{i=1}^{i=p}(G_i \mapsto \{\ \langle vl_i \rangle\ \})) \vdash B \Rightarrow \mathcal{C}', tl \\ \mathcal{C} \oplus \mathcal{C}_1 \vdash B_1 \Rightarrow \mathcal{C}', tl' \quad \ldots \quad \mathcal{C} \oplus \mathcal{C}_n \vdash B_n \Rightarrow \mathcal{C}', tl'\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\textbf{trap}\ G_1\ \langle VL_1 \rangle\ \rightarrow B_1\ ...\ G_n\ \langle VL_n \rangle\ \rightarrow B_n\ \textbf{in}\ B\ \textbf{endtrap}) \\ \Rightarrow \mathcal{C}', tl\end{array}} [\delta \notin \{G_1, .., G_n\}] \tag{44}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash VL_1 \Rightarrow vl_1, \mathcal{C}_1, \{\} \quad \ldots \mathcal{C} \vdash VL_n \Rightarrow vl_n, \mathcal{C}_n, \{\} \\ \mathcal{C} \oplus (+_{i=1}^{i=p}(G_i \mapsto \{\ \langle vl_i \rangle\ \})) \vdash B \Rightarrow \mathcal{C}', tl' \\ \mathcal{C}' \subset \{\}\ \langle \mathcal{C}' \subset \mathcal{C}_1 \rangle \\ \mathcal{C} \oplus \mathcal{C}_1 \vdash B_1 \Rightarrow \mathcal{C}'', tl'' \quad \ldots \quad \mathcal{C} \oplus \mathcal{C}_n \vdash B_n \Rightarrow \mathcal{C}'', tl''\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\textbf{trap}\ G_1\ \langle VL_1 \rangle\ \rightarrow B_1 ... G_n\ \langle VL_n \rangle\ \rightarrow B_n\ \textbf{in}\ B\ \textbf{endtrap}) \\ \Rightarrow \mathcal{C}'', tl''\end{array}} [\delta = G_1] \tag{45}$$

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathcal{C}_1, () \qquad\qquad \mathcal{C} \oplus \mathcal{C}_i \vdash B_2 \Rightarrow \mathcal{C}_2, tl}{\mathcal{C} \vdash (B_1\ \ ;\ \ B_2) \Rightarrow \mathcal{C}_2, tl} \tag{46}$$

$$\frac{\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow \mathcal{C}'}{\mathcal{C} \vdash (P\ \texttt{:=}\ \textbf{any}\ T) \Rightarrow \mathcal{C}', ()} \tag{47}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \Rightarrow \mathcal{C}', tl \\ \mathcal{C} \vdash B_1 \Rightarrow \mathcal{C}', tl\end{array}}{\mathcal{C} \vdash (B_1\, \texttt{[>} B_2) \Rightarrow \mathcal{C}', tl} \tag{48}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash G_1 \Rightarrow cht_1 \quad \ldots \quad \mathcal{C} \vdash G_p \Rightarrow cht_p \\ \Sigma_1 \subset \{0 \ldots n\} \quad \ldots \quad \Sigma_p \subset \{0 \ldots n\} \\ \mathcal{C} \vdash B_0 \Rightarrow \mathcal{C}_0, tl \quad \ldots \quad \mathcal{C} \vdash B_n \Rightarrow \mathcal{C}_n, tl\end{array}}{\mathcal{C} \vdash \left(\begin{array}{l}\textbf{par}\ G_1 \!:\! \Sigma_1 \ldots G_p \!:\! \Sigma_p \\ \quad B_0\,||\ldots||\,B_n \\ \textbf{endpar}\end{array}\right) \Rightarrow +_{i=1}^{i=n}\mathcal{C}_i, tl} \tag{49}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow T_1' \quad \ldots \quad \mathcal{C} \vdash T_n \Rightarrow T_n' \\ \mathcal{C} \oplus (+_{i=1}^{i=n} V_i \mapsto (T_i, def)) \vdash B_0 \Rightarrow \mathcal{C}_0, tl\end{array}}{\mathcal{C} \vdash (\textbf{par}\ V_1 \!:\! T_1 \ldots V_n \!:\! T_n\ \ ||\,B_0\ \textbf{endpar}) \Rightarrow \mathcal{C}_0, tl} \tag{50}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash M_0 \Rightarrow \mathcal{C}_0 \quad \mathcal{C} \oplus \mathcal{C}_0 \vdash B_0 \Rightarrow \mathcal{C}', tl \quad \ldots \\ \mathcal{C} \vdash M_p \Rightarrow \mathcal{C}_p \quad \mathcal{C} \oplus \mathcal{C}_p \vdash B_p \Rightarrow \mathcal{C}', tl\end{array}}{\mathcal{C} \vdash (\textbf{case}\ M_0 \rightarrow B_0 \ldots M_p \rightarrow B_p\ \textbf{endcase}) \Rightarrow \mathcal{C}', tl} \tag{51}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow T_1' \quad \ldots \quad \mathcal{C} \vdash T_n \Rightarrow T_n' \\ \mathcal{C} \oplus (+_{i=1}^{i=n} V_i \mapsto (T_i', undef)) \vdash B \Rightarrow \mathcal{C}', tl\end{array}}{\mathcal{C} \vdash (\textbf{local}\ V_1 \!:\! T_1, \ldots, V_n \!:\! T_n\ \textbf{in}\ B\ \textbf{endloc}) \Rightarrow \mathcal{C}' \ominus \{V_1, \ldots, V_n\}, tl} \tag{52}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow T_1' \quad \ldots \quad \mathcal{C} \vdash T_n \Rightarrow T_n' \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, T_1' \quad \ldots \quad \mathcal{C} \vdash E_n \Rightarrow \{\}, T_n' \\ \mathcal{C} \oplus (+_{j=1}^{j=n} V_j \mapsto (T_j', def)) \vdash B_0 \Rightarrow \mathcal{C}', T'\end{array}}{\mathcal{C} \vdash \left(\begin{array}{l}\textbf{loop } V_1 \!:\! T_1 \!:=\! E_1, \ldots, V_n \!:\! T_n \!:=\! E_n \\ \textbf{in } B_0 \\ \textbf{endloop}\end{array}\right) \Rightarrow \mathcal{C}', T'} \tag{53}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash V_1 \Rightarrow (T_1, def) \quad \ldots \quad \mathcal{C} \vdash V_n \Rightarrow (T_n, def) \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, T_1 \quad \ldots \quad \mathcal{C} \vdash E_n \Rightarrow \{\}, T_n\end{array}}{\mathcal{C} \vdash (\textbf{continue} V_1 \!:=\! E_1, \ldots, V_n \!:=\! E_n) \Rightarrow \{\}, ()} \tag{54}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \Gamma_1 \Rightarrow cht_1 \quad \ldots \quad \mathcal{C} \vdash \Gamma_n \Rightarrow cht_n \\ \mathcal{C} \oplus (+_{i=1}^{i=n} G_1 \mapsto cht_i) \vdash B_0 \Rightarrow \mathcal{C}_0, tl\end{array}}{\mathcal{C} \vdash (\textbf{hide } G_1 \!:\! \Gamma_1, \ldots, G_n \!:\! \Gamma_n \textbf{ in } B_0) \Rightarrow \mathcal{C}_0, tl} \tag{55}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \Pi \Rightarrow (G_1 : cht_1, \ldots, G_p : cht_p) \rightarrow (V_1 : T_1 : A_1, \ldots, V_n : T_n : A_n) \rightarrow tl \\ \mathcal{C} \vdash G_1' \Rightarrow cht_1 \quad \ldots \quad \mathcal{C} \vdash G_p' \Rightarrow cht_p \\ \mathcal{C} \vdash E_1 \Rightarrow \{\}, (T_1) \quad | \quad \mathcal{C} \vdash (P_1 \Rightarrow T_1) \Rightarrow \mathcal{C}_1 \quad \ldots \\ \mathcal{C} \vdash E_n \Rightarrow \{\}, (T_n) \quad | \quad \mathcal{C} \vdash (P_n \Rightarrow T_n) \Rightarrow \mathcal{C}_n\end{array}}{\begin{array}{c}\mathcal{C} \vdash (\Pi\,[G_1', \ldots, G_p']\,(E_1 \mid \& P_1, \ldots, E_n \mid \& P_n)) \Rightarrow \\ (\{\} \mid \mathcal{C}_1) + \ldots + (\{\} \mid \mathcal{C}_n), tl\end{array}} \tag{56}$$

$$\frac{\begin{array}{c}\mathcal{C}(G_0') \ni \{(V_1'^0 : T_1'^0 : A_1^0, \ldots, V_{m_0}'^0 : T_{m_0}'^0 : A_{m_0}^0)\} \quad \ldots \\ \mathcal{C}(G_p') \ni \{(V_1'^p : T_1'^p : A_1^p, \ldots, V_{m_p}'^p : T_{m_p}'^p : A_{m_p}^p)\} \\ \mathcal{C} \vdash VL_0 \Rightarrow vl_0, \mathcal{C}_0, \{\} \quad \ldots \quad \mathcal{C} \vdash VL_p \Rightarrow vl_p, \mathcal{C}_p, \{\} \\ \mathcal{C} \oplus \mathcal{C}_0 \vdash E_1'^0 \Rightarrow \{\}, (T_1'^0) \quad \ldots \quad \mathcal{C} \oplus \mathcal{C}_p \vdash E_{m_p}'^p \Rightarrow \{\}, (T_{m_p}'^p) \\ \mathcal{C} \oplus (+_{i=0}^{i=n} G_i \mapsto \{vl_i\}) \vdash B \Rightarrow \mathcal{C}', tl\end{array}}{\mathcal{C} \vdash \left(\begin{array}{l}\textbf{rename} \\ G_0 \langle VL_0 \rangle \;\; \textbf{:=} \;\; G_0' \langle (E_1'^0, \ldots, E_{m_0}'^0) \rangle \\ \ldots \\ \quad G_p \langle VL_p \rangle \;\; \textbf{:=} \;\; G_p' \langle (E_1'^p, \ldots, E_{m_p}'^p) \rangle \\ \textbf{in } B \\ \textbf{endren}\end{array}\right) \Rightarrow \mathcal{C}', tl} \tag{57}$$

4.8 Translation function from reduced user language to minimal core language

The functionality of expressions and behaviour, denoted by $\mathcal{F}$, is an application from Context $\times (\mathcal{E}_r \cup \mathcal{B}_r)$ to VarEnv $\times$ TuplType, defined at the static semantics level, by the static semantics rules:

- for an expression E, if $\mathcal{C} \vdash E \Rightarrow \mathcal{C}', tl$ then $\mathcal{F}(E) \stackrel{\text{def}}{=} \mathcal{C}', tl$. For instance,

$$\begin{array}{rcl}\mathcal{F}(K) & \stackrel{\text{def}}{=} & \{\}, (T) \;\; where \; [K : T] \\ \mathcal{F}(V) & \stackrel{\text{def}}{=} & \{\}, (T) \;\; where \; \mathcal{C}(V) = (T, def) \\ \mathcal{F}(P \!:=\! E) & \stackrel{\text{def}}{=} & \mathcal{C}', ()\end{array}$$

$$\textit{where } \mathcal{C} \vdash E \Rightarrow \{\}, (T) \textbf{ and}$$
$$\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow \mathcal{C}'$$

- for a behaviour expression B, if $\mathcal{C} \vdash B \Rightarrow \mathcal{C}', tl$, then $\mathcal{F}(B) \stackrel{\texttt{def}}{=} \mathcal{C}', tl$. From static semantics rules we have:

$$\begin{array}{rcl} \mathcal{F}(noexit) & \stackrel{\texttt{def}}{=} & \{\}, () \\ \mathcal{F}(exit(()T_1, \ldots T_n)) & \stackrel{\texttt{def}}{=} & \{\}, (T_1, \ldots, T_n) \\ \mathcal{F}(P\texttt{:=}E) & \stackrel{\texttt{def}}{=} & \mathcal{C}', () \\ & & \textit{where } \mathcal{C} \vdash E \Rightarrow \{\}, (T) \textbf{ and} \\ & & \quad \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow \mathcal{C}' \end{array}$$

The translation functions from reduced user language to minimal core language uses static semantics and functionality function. It supposes to translate the reduced language phrases where overloading is solved by static semantics, and to translate them into core language phrases as define in ??:

$$\begin{array}{rl} \ll .,. \gg : & (\text{Context} \rightarrow \mathcal{D}_r \rightarrow \mathcal{D}_m) \cup \\ & (\text{Context} \rightarrow \mathcal{P}_r \rightarrow \mathcal{P}_m) \cup \\ & (\text{Context} \rightarrow \mathcal{M}_r \rightarrow \mathcal{M}_m) \cup \\ & (\text{Context} \rightarrow \mathcal{E}_r \rightarrow (\mathcal{E}_m \cup \mathcal{B}_m)) \cup \\ & (\text{Context} \rightarrow \mathcal{B}_r \rightarrow \mathcal{B}_m) \end{array}$$

4.8.1 Translation of declarations

The translation function for declarations is an identity for type, and channel declarations.

Exception declaration: exception types are a special case of channel, So:

$$\ll \textbf{exception } \Xi \textbf{ is } (V_1\!:\!T_1, ..., V_n\!:\!T_n) \textbf{ endexc} \gg$$
$$\stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{channel } \Xi \textbf{ is} \\ \quad (V_1\!:\!T_1, ..., V_n\!:\!T_n) \\ \textbf{endchan} \end{array} \right)$$

Function declaration: is a special case of process declaration. So:

$$\left\langle\!\left\langle \left(\begin{array}{l} \textbf{function } F\,\texttt{[}X_1\!:\!\Xi_1, \ldots, X_p\!:\!\Xi_p\texttt{]} \\ \qquad\qquad (\textbf{in} \mid \textbf{out } V_1\!:\!T_1, ..., \textbf{in} \mid \textbf{out } V_n\!:\!T_n) \\ \qquad [:T] \textbf{ is} \\ \quad E \\ \textbf{endfunc } [F] \end{array} \right) \right\rangle\!\right\rangle$$
$$\stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{process } F\,\texttt{[}X_1\!:\!\Xi_1, \ldots, X_p\!:\!\Xi_p\texttt{]} \\ \qquad\qquad (\textbf{in} \mid \textbf{out } V_1\!:\!T_1, ..., \textbf{in} \mid \textbf{out } V_n\!:\!T_n) \\ \qquad [\textbf{:noexit} \mid \textbf{:exit } (T)] \textbf{ is} \\ \quad \ll E \gg \\ \textbf{endproc} \end{array} \right)$$

and the function binding is unique by overloading solving.

4.8.2 Translation of patterns

The translation function for patterns is an identity for all pattern phrases of reduced language, except layered patterns P **of** T which are translated in P, because type is no more needed.

4.8.3 Translation of match expressions

The translation function for patterns is an identity for all match expressions. For instance,

$$\ll E::P \gg \stackrel{\mathtt{def}}{=} \ll E \gg::\ll P \gg$$

$$\ll M \textbf{ when } E \gg \stackrel{\mathtt{def}}{=} \ll M \gg \textbf{when} \ll E \gg$$

4.8.4 Translation of value expressions

Values expressions of reduced language are translated in values terms or in behaviour expressions of minimal language.

Value terms:

$$\ll K \gg \stackrel{\mathtt{def}}{=} \delta(K)$$

$$\ll V \gg \stackrel{\mathtt{def}}{=} \delta(V)$$

$$\begin{array}{l} \ll C(E_1, \ldots, E_n) \gg \\ \stackrel{\mathtt{def}}{=} \left(\begin{array}{l} \textbf{trap } \delta(V_1 : T_1) \rightarrow \\ \ldots \\ \qquad \textbf{trap } \delta(V_n : T_n) \rightarrow \delta(C(V_1, \ldots, V_n)) \\ \qquad \textbf{in} \ll E_n \gg \textbf{endtrap} \\ \ldots \\ \textbf{in} \ll E_1 \gg \textbf{endtrap} \end{array} \right) \\ \quad \textit{where } \mathcal{C} \vdash E_i \Rightarrow \mathcal{C}_i, (T_i) \end{array}$$

Function call:

$$\ll F\mathtt{[}GL\mathtt{]}(\ldots E_i \ldots \mathtt{\&}P_j \ldots) \gg$$

$$\stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \ldots \\ \quad \textbf{trap } \delta(V_i : T_i) \rightarrow \\ \ldots \\ \qquad\qquad F\,\texttt{[}GL\texttt{]}\texttt{(}\ldots V_i \ldots \texttt{\&}P_j \ldots\texttt{)} \\ \ldots \\ \quad \textbf{in} \ll E_i \gg \textbf{endtrap} \\ \ldots \\ \textbf{endtrap} \end{array} \right)$$

$$where\ \mathcal{C} \vdash E_i \Rightarrow \mathcal{C}_i, (T_i)$$

Case expressions: the translation function is an identity morphism.

Raise exception:

$$\ll \textbf{raise } X\texttt{(}E_1, ..., E_n\texttt{)} \gg$$

$$\stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{trap } \delta(V_1 : T_1) \rightarrow \\ \ldots \\ \qquad\qquad \textbf{trap } \delta(V_n : T_n) \rightarrow X\texttt{(}V_1, \ldots, V_n\texttt{)} \\ \qquad\qquad \textbf{in} \ll E_n \gg \textbf{endtrap} \\ \ldots \\ \textbf{in} \ll E_1 \gg \textbf{endtrap} \end{array} \right)$$

$$where\ \mathcal{C} \vdash E_i \Rightarrow \mathcal{C}_i, (T_i)$$

Trap exceptions:

$$\left\langle\!\left\langle \left(\begin{array}{l} \textbf{trap} \\ \quad X_1\texttt{(}V_1^1 : T_1^1, ..., V_{p_1}^1 : T_{p_1}^1\texttt{)} \rightarrow E_1 \\ \quad ... \\ \quad X_n\texttt{(}V_1^n : T_1^n, ..., V_{p_n}^n : T_{p_n}^n\texttt{)} \rightarrow E_n \\ \textbf{in } E \\ \textbf{endtrap} \end{array} \right) \right\rangle\!\right\rangle$$

$$\stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \quad X_1\texttt{(}V_1^1 : T_1^1, ..., V_{p_1}^1 : T_{p_1}^1\texttt{)} \rightarrow \ll E_1 \gg \\ \quad ... \\ \quad X_n\texttt{(}V_1^n : T_1^n, ..., V_{p_n}^n : T_{p_n}^n\texttt{)} \rightarrow \ll E_n \gg \\ \textbf{in} \ll E \gg \\ \textbf{endtrap} \end{array} \right)$$

Local declaration:

$$\ll \textbf{local } V_1 : T_1, \ldots, V_n : T_n \textbf{ in } E \textbf{ endloc} \gg$$

$$\stackrel{\texttt{def}}{=} \quad \textbf{local } V_1 : T_1, \ldots, V_n : T_n \textbf{ in} \ll E \gg \textbf{endloc}$$

Variable assignment:

$$\begin{array}{l} \ll P\,\texttt{:=}E \gg \\ \stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{trap } \delta(V:T) \rightarrow \textbf{case } V::P \rightarrow \delta(\mathcal{F}(P)) \\ \textbf{in} \ll E \gg \textbf{endtrap} \end{array} \right) \\ \quad \textit{where } \mathcal{C} \vdash E \Rightarrow \mathcal{C}', (T) \end{array}$$

Expression sequencing:

$$\ll E_1 \;\texttt{;}\; E_2 \gg \stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \quad \delta\mathcal{F}(E_1) \rightarrow \ll E_2 \gg \\ \textbf{in} \ll E_1 \gg \textbf{endtrap} \end{array} \right)$$

Expression equality:

$$\ll E_1 \;\texttt{=}\; E_2 \gg \stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{trap } \delta(V_1:T_1) \rightarrow \\ \qquad\qquad \textbf{trap} \\ \qquad\qquad\quad \delta(V_2:T_2) \rightarrow V_1 \;\texttt{=}\; V_2 \\ \qquad\qquad \textbf{in} \ll E_2 \gg \textbf{endtrap} \\ \textbf{in} \ll E_1 \gg \textbf{endtrap} \end{array} \right)$$
$$\textit{where } \mathcal{C} \vdash E_i \Rightarrow \{\}, T_i$$

Select expression:

$$\ll E_0\texttt{.}V_0 \gg \stackrel{\texttt{def}}{=} \left(\begin{array}{l} \textbf{trap}\delta(V:T_0) \rightarrow \\ \qquad\quad \textbf{case} \\ \qquad\qquad V::C_1(\ldots V_0 \ldots) \rightarrow \delta(V_0) \\ \qquad\qquad \ldots \\ \qquad\qquad V::C_p(\ldots V_0 \ldots) \rightarrow \delta(V_0) \\ \qquad\quad \textbf{endcase} \\ \textbf{in} \ll E_0 \gg \textbf{endtrap} \end{array} \right)$$
$$\textit{where } \mathcal{C} \vdash E_0 \Rightarrow \{\}, T_0$$

where $C_1, \ldots, C_p$ are the constructors of type of E_0 having V_0 as field (they exists by the static semantics check).

Field update expression:

$$\ll E_0\texttt{.}\{V_1\texttt{:=}E_1, ..., V_n\texttt{:=}E_n\} \gg$$

$$
\stackrel{\text{def}}{=}
\left(
\begin{array}{l}
\textbf{trap } \delta(V_0 : T_0) \rightarrow \\
\qquad \textbf{case} \\
\qquad\quad V_0\texttt{::}C_1\ VL_1 \rightarrow \ll\ V_1'\texttt{:=}E_1\ \ ;\ \ \ldots\ \ ;\ \ V_n'\texttt{:=}E_n\ \ ;\ \ \delta(C_1\ VL_1')\ \gg \\
\qquad\quad \ldots \\
\qquad\quad V_0\texttt{::}C_p\ VL_p \rightarrow \ll\ V_1'\texttt{:=}E_1\ \ ;\ \ \ldots\ \ ;\ \ V_n'\texttt{:=}E_n\ \ ;\ \ \delta(C_p\ VL_p')\ \gg \\
\qquad \textbf{endcase} \\
\textbf{in} \ll\ E_0\ \gg \textbf{endtrap}
\end{array}
\right)
$$

$$
\textit{where } \mathcal{C} \vdash E_i \Rightarrow \{\}, T_i
$$

where $C_1, \ldots, C_p$ are the constructors of type of E_0 having $V_1, \ldots, V_n$ as field (they exists by the static semantics check). The new variable lists VL_i' are built from old values of parameters iff they are not affected and the new values $V_1', \ldots, V_n'$.

Loop expression:

$$
\left\langle\!\!\left\langle
\left(
\begin{array}{l}
\textbf{loop} \\
\quad V_1\texttt{:}T_1\texttt{:=}E_1, \ldots, V_n\texttt{:}T_n\texttt{:=}E_n \textbf{ in} \\
\textbf{in } E_0 \\
\textbf{endproc}
\end{array}
\right)
\right\rangle\!\!\right\rangle
$$

$$
\begin{array}{ll}
\stackrel{\text{def}}{=} & \ll\ \Pi\texttt{[}exc(E_0)\texttt{]}\texttt{(}E_1, \ldots, E_n\texttt{)}\ \gg \\
& \textit{where} \\
& \textbf{process } \Pi\texttt{[}exc(E_0)\texttt{]}\texttt{(}\textbf{in}V_1\texttt{:}T_1, \ldots, \textbf{in}V_n\texttt{:}T_n\texttt{)} \\
& \qquad\quad \texttt{:}\textbf{exit } (\mathcal{F}_{tl}(E_0)) \textbf{ is} \\
& \quad \textbf{trap } \xi\texttt{(}V_1'\texttt{:}T_1, \ldots, V_n'\texttt{:}T_n\texttt{)} \rightarrow \Pi\texttt{[}exc(E_0)\texttt{]}\texttt{(}V_1', \ldots, V_n'\texttt{)} \\
& \quad \textbf{in} \ll\ E_0\ \gg \\
& \quad \textbf{endtrap} \\
& \textbf{endproc}
\end{array}
$$

where $exc(E_0)$ is the set of exceptions, different from ξ raised by E_0.

Loop continue:

$$
\begin{array}{c}
\ll\ \textbf{continue } \texttt{(}V_1\texttt{:=}E_1 \ldots, V_n\texttt{:=}E_n\texttt{)}\ \gg \\
\stackrel{\text{def}}{=}\quad \ll\ \xi\texttt{(}V_1\texttt{:=}E_1 \ldots V_n\texttt{:=}E_n\texttt{)}\ \gg
\end{array}
$$

4.8.5 Translation of behaviour expressions

Behaviour expressions of reduced language are translated in behaviour expressions of minimal language.

Action:

$$
\ll\ G\texttt{(}E_1 \mid \texttt{\&}P_1, \ldots, E_n \mid \texttt{\&}P_n\texttt{)}\ \gg
$$

$$\stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap}\ \delta(V_1 : T_1) \rightarrow \\ \ldots \\ \qquad \textbf{trap}\ \delta(V_n : T_n) \rightarrow G(V_1 \mid \&P_1, \ldots, V_n \mid \&P_n) \\ \qquad \textbf{in} \ll E_n \gg \textbf{endtrap} \\ \ldots \\ \textbf{in} \ll E_1 \gg \textbf{endtrap} \end{array} \right)$$

$$\textit{where}\ \mathcal{C} \vdash E_i \Rightarrow \mathcal{C}_i, (T_i)$$

Assignment behaviour expression:

$$\ll P := E \gg$$
$$\stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \quad \delta(V : T) \rightarrow \textbf{case}\ V :: P \rightarrow \delta(\mathcal{F}(P))\ \textbf{endcase} \\ \textbf{in} \ll E \gg \textbf{endtrap} \end{array} \right)$$
$$\textit{where}\ \mathcal{C} \vdash E \Rightarrow \mathcal{C}', (T)$$

Trap expression: the translation function is an identity morphism.

Sequential composition:

$$\ll B_1 ; B_2 \gg$$
$$\stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap}\ \delta\mathcal{F}(B_1) \rightarrow \ll B_2 \gg \\ \textbf{in} \\ \quad \ll B_1 \gg \\ \textbf{endtrap} \end{array} \right)$$

Choice values: the translation function is an identity morphisms.

Disabling:

$$\ll B_1 [> B_2 \gg$$
$$\stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap} \quad \theta \rightarrow \ll B_2 \gg \\ \textbf{in} \\ \qquad \textbf{par} \\ \qquad\quad \ll B_1 \gg || \\ \qquad\quad \textbf{trap}\ \delta(V : \textbf{enum}(1,2)) \rightarrow \\ \qquad\qquad \textbf{case}\ V :: 1 \rightarrow \delta\mathcal{F}(B_1) \\ \qquad\qquad\qquad V :: 2 \rightarrow X \\ \qquad\qquad \textbf{endcase} \\ \qquad\quad \textbf{in}\ V := \textbf{any}\ \textbf{enum}(1,2) \\ \qquad\quad \textbf{endtrap} \\ \qquad \textbf{endpar} \\ \textbf{endtrap} \end{array} \right)$$

For general parallel composition, parallel over values, general case behaviour and local declaration behaviour, the translation function is an identity morphism.

Loop behaviour:

$$\left\langle\!\!\left\langle \left(\begin{array}{l} \textbf{loop } V_1\!:\!T_1\!:=\!E_1,\ldots,V_n\!:\!T_n\!:=\!E_n \textbf{ in} \\ \textbf{in } B_0 \\ \textbf{endproc} \end{array} \right) \right\rangle\!\!\right\rangle$$

$$\begin{array}{ll} \stackrel{\text{def}}{=} & \ll \Pi\texttt{[}gate(B_0)\texttt{]}\texttt{(}E_1,\ldots,E_n\texttt{)} \gg \\ & where \\ & \textbf{process } \Pi\texttt{[}gate(B_0)\texttt{]}\texttt{(}\textbf{in}V_1\!:\!T_1,\ldots,\textbf{in}V_n\!:\!T_n\texttt{)}\texttt{:}\textbf{exit } (\mathcal{F}_{tl}(B_0)) \textbf{ is} \\ & \quad \textbf{trap } \xi\texttt{(}V_1'\!:\!T_1,\ldots,V_n'\!:\!T_n\texttt{)} \rightarrow \Pi\texttt{[}gate(B_0)\texttt{]}\texttt{(}V_1',\ldots,V_n'\texttt{)} \\ & \quad \textbf{in} \ll B_0 \gg \\ & \quad \textbf{endtrap} \\ & \textbf{endproc} \end{array}$$

where $gate(B_0)$ is the set of gates, different from ξ raised by B_0.

Loop continue:

$$\begin{array}{l} \ll \textbf{continue } \texttt{(}V_1\!:=\!E_1\ldots,V_n\!:=\!E_n\texttt{)} \gg \\ \quad \stackrel{\text{def}}{=} \quad \ll \xi\texttt{(}V_1\!:=\!E_1\ldots V_n\!:=\!E_n\texttt{)} \gg \end{array}$$

Hiding: the translation function is an identity morphism.

Process instantiation:

$$\begin{array}{ll} \multicolumn{2}{l}{\ll \Pi\texttt{[}GL\texttt{]}\texttt{(}E_1 \mid \texttt{\&}P_1,\ldots,E_n \mid \texttt{\&}P_n\texttt{)} \gg} \\ \stackrel{\text{def}}{=} & \left(\begin{array}{l} \textbf{trap } \delta(V_1\!:\!T_1) \rightarrow \\ \ldots \\ \qquad\qquad \textbf{trap } \delta(V_n\!:\!T_n) \rightarrow \Pi\texttt{[}GL\texttt{]}\texttt{(}V_1 \mid \texttt{\&}P_1,\ldots,V_n \mid \texttt{\&}P_n\texttt{)} \\ \qquad\qquad \textbf{in} \ll E_n \gg \textbf{endtrap} \\ \ldots \\ \textbf{in} \ll E_1 \gg \textbf{endtrap} \end{array} \right) \\ & where\ \mathcal{C} \vdash E_i \Rightarrow \mathcal{C}_i, (T_i) \end{array}$$

Post-renaming: the translation function is an identity morphism up to translation of expressions as values terms.

4.9 Dynamic semantics of the minimal core language

4.9.1 Environments

An environment is a multi-set of bindings:

$\mathcal{E}$	$::=$	$\Pi \mapsto (\lambda GL.VL.B)$	*procesbinding*
	$\mid$	$C \mapsto (VL \mapsto T)$	*constructor environment*
	$\mid$	$\{\}$	*empty environment*
	$\mid$	$\mathcal{E}, \mathcal{E}$	*composition of environments*
v	$::=$	K	*value special constants*
	$\mid$	$C(v_1, \ldots, v_n)$	*constrycted value*
vl	$::=$	$()$	*emptylist*
	$\mid$	$(v_1, \ldots, v_n)$	*list of values*

4.9.2 Dynamic semantics of variable list

The dynamic semantics is given by assertion: $\mathcal{E} \vdash (VL \Rightarrow vl) \Rightarrow \sigma \mid \textbf{fail}$ meaning "In the environment $\mathcal{E}$, the variable list VL matched over the value list vl gives the substitution σ or **fail**s to match."

$$\frac{}{\mathcal{E} \vdash () \Rightarrow () \Rightarrow []} \qquad (1)$$

$$\frac{}{\mathcal{E} \vdash ((V_1 : T_1, \ldots, V_n : T_n) \Rightarrow (v_1, \ldots, v_n)) \Rightarrow [v_1/V_1, \ldots, v_n/V_n]} \qquad (2)$$

4.9.3 Dynamic semantics of declarations

The dynamic semantics is given by assertion: $\mathcal{E} \vdash D \Rightarrow \mathcal{E}'$ meaning "In the environment $\mathcal{E}$, the declaration D is well formed and gives the environment $\mathcal{E}$'."

$$\frac{}{\begin{array}{c}\mathcal{E} \vdash (\textbf{type } T \textbf{ is } C_1\ VL_1 \ldots C_p\ VL_p \textbf{ endtype}) \Rightarrow \\ C_1 \mapsto VL_1 \rightarrow T, \ldots, C_p \mapsto VL_p \rightarrow T\end{array}} \qquad (3)$$

$$\frac{}{\begin{array}{c}\mathcal{E} \vdash (\textbf{process } \Pi[GL]VL \,\langle \textbf{: exit } (T_1, ..., T_m)\rangle \textbf{ is } B \textbf{ endproc}) \Rightarrow \\ (P \mapsto GL \rightarrow VL \rightarrow (\ \langle T_1', \ldots, T_m'\rangle\ ) \rightarrow B)\end{array}} \qquad (4)$$

4.9.4 Dynamic semantics of patterns

The dynamic semantics is given by assertion: $\mathcal{E} \vdash (P \Rightarrow v) \Rightarrow \sigma \mid \textbf{fail}$ meaning "In the environment $\mathcal{E}$, the pattern P matched over the value v gives the substitution σ or **fail**s to match."

$$\frac{}{\mathcal{E} \vdash (K \Rightarrow K) \Rightarrow []} \tag{5}$$

$$\frac{}{\mathcal{E} \vdash (K \Rightarrow K') \Rightarrow \textbf{fail}} \quad [K \neq K'] \tag{6}$$

$$\frac{}{\mathcal{E} \vdash (V \Rightarrow v) \Rightarrow [v/V]} \tag{7}$$

$$\frac{}{\mathcal{E} \vdash (\textbf{any}\ T \Rightarrow v) \Rightarrow ()} \tag{8}$$

$$\frac{\mathcal{E} \vdash (P_i \Rightarrow v_i) \Rightarrow \sigma_i \mid \textbf{fail}}{\mathcal{E} \vdash (C(P_1, ..., P_n) \Rightarrow C(v_1, \ldots, v_n)) \Rightarrow (\sigma_1 \mid \textbf{fail}) \circ \ldots \circ (\sigma_n \mid \textbf{fail})} \tag{9}$$

$$\frac{}{\mathcal{E} \vdash (C(P_1, ..., P_n) \Rightarrow v) \Rightarrow \textbf{fail}} \quad [v \neq C(\ldots)] \tag{10}$$

4.9.5 Dynamic semantics of match expressions

The dynamic semantics is given by assertion: $\mathcal{E} \vdash M \Rightarrow \sigma \mid \textbf{fail}$ meaning "In the environment $\mathcal{E}$, the match expression M gives the substitution σ or **fail**."

$$\frac{\begin{array}{c}\mathcal{E} \vdash E \Rightarrow v \\ \mathcal{E} \vdash (P \Rightarrow v) \Rightarrow \sigma \mid \textbf{fail}\end{array}}{\mathcal{E} \vdash M \Rightarrow \sigma \mid \textbf{fail}} \tag{11}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash M \Rightarrow \sigma \\ \mathcal{E} \vdash E[\sigma] \Rightarrow \mathit{true}\end{array}}{\mathcal{E} \vdash (M\ \textbf{when}\ E) \Rightarrow \sigma} \tag{12}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash M \Rightarrow \sigma \\ \mathcal{E} \vdash E[\sigma] \Rightarrow \mathit{false}\end{array}}{\mathcal{E} \vdash (M\ \textbf{when}\ E) \Rightarrow \textbf{fail}} \tag{13}$$

$$\frac{\mathcal{E} \vdash M \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (M\ \textbf{when}\ E) \Rightarrow \textbf{fail}} \tag{14}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash M_1 \Rightarrow \sigma_1 \\ \mathcal{E} \vdash M_2 \Rightarrow \sigma_2\end{array}}{\mathcal{E} \vdash (M_1 \textbf{ and } M_2) \Rightarrow \sigma_1 \circ \sigma_2} \tag{15}$$

$$\frac{\mathcal{E} \vdash M_1 \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (M_1 \textbf{ and } M_2) \Rightarrow \textbf{fail}} \tag{16}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash M_1 \Rightarrow \sigma_1 \\ \mathcal{E} \vdash M_2 \Rightarrow \textbf{fail}\end{array}}{\mathcal{E} \vdash (M_1 \textbf{ and } M_2) \Rightarrow \textbf{fail}} \tag{17}$$

$$\frac{\mathcal{E} \vdash M_1 \Rightarrow \sigma_1}{\mathcal{E} \vdash (M_1 \textbf{ or } M_2) \Rightarrow \sigma_1} \tag{18}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash M_1 \Rightarrow \textbf{fail} \\ \mathcal{E} \vdash M_2 \Rightarrow \sigma_2\end{array}}{\mathcal{E} \vdash (M_1 \textbf{ or } M_2) \Rightarrow \sigma_2} \tag{19}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash M_1 \Rightarrow \textbf{fail} \\ \mathcal{E} \vdash M_2 \Rightarrow \textbf{fail}\end{array}}{\mathcal{E} \vdash (M_1 \textbf{ or } M_2) \Rightarrow \textbf{fail}} \tag{20}$$

4.9.6 Dynamic semantics of expressions

The dynamic semantics is given by assertion: $\mathcal{E} \vdash E \Rightarrow v$ meaning "In the environment $\mathcal{E}$, the expression E gives the value v."

$$\frac{}{\mathcal{E} \vdash K \Rightarrow K} \tag{21}$$

$$\frac{}{\mathcal{E} \vdash V \Rightarrow V} \tag{22}$$

$$\frac{\mathcal{E} \vdash E_i \Rightarrow v_i}{\mathcal{E} \vdash C(E_1, \ldots, E_n) \Rightarrow C(v_1, \ldots, v_n)} \tag{23}$$

$$\frac{\mathcal{E} \vdash E_i \Rightarrow v_i}{\mathcal{E} \vdash E_1 \texttt{ = } E_2 \Rightarrow v_1 = v_2} \tag{24}$$

4.9.7 Dynamic semantics of behaviours

The dynamic semantics is given by assertion: $\mathcal{E} \vdash \mathcal{E} \vdash B \xrightarrow{l} B'$ meaning "In the environment $\mathcal{E}$, the behaviour B makes the transition labeled l and gives the behaviour B'." The form of labels is $l = (G|\delta)\ v_1, \ldots v_n$.

Action:

$$\frac{\begin{array}{c}\mathcal{E} \vdash (P_i \Rightarrow v_i) \Rightarrow \sigma_i \quad | \quad \mathcal{E} \vdash E_i \Rightarrow v_i \\ \mathcal{E} \vdash E[\circ_i \sigma_i] \Rightarrow \mathit{true}\end{array}}{\mathcal{E} \vdash \mathcal{E} \vdash G(E_1 \mid \&P_1, \ldots, E_n \mid \&P_n)\ [[E]] \xrightarrow{G\ v_1 \ldots v_n} \delta\ v_1 \ldots v_n} \tag{25}$$

Trap gates:

$$\frac{\mathcal{E} \vdash B_0 \xrightarrow{l} B'_0}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{trap} \\ \quad \{\, G_i V L_i \rightarrow B_i \,\}_{i \in 1..n} \\ \textbf{in } B_0 \\ \textbf{endtrap}\end{array}\right) \xrightarrow{l} \left(\begin{array}{l}\textbf{trap} \\ \quad \{\, G_i V L_i \rightarrow B_i \,\}_{i \in 1..n} \\ \textbf{in } B'_0 \\ \textbf{endtrap}\end{array}\right)} \tag{26}$$

$$\mathit{gate}(l) \notin \{G_1, ..G_n\} \tag{27}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_0 \xrightarrow{G\ v_1,\ldots v_p} B'_0 \\ \mathcal{E} \vdash V L_j \Rightarrow (v_1, \ldots v_p) \Rightarrow \sigma \\ \mathcal{E} \vdash B_j[\sigma] \xrightarrow{l} B'_j\end{array}}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{trap} \\ \quad \{\, G_i V L_i \rightarrow B_i \,\}_{i \in 1..n} \\ \textbf{in } B_0 \\ \textbf{endtrap}\end{array}\right) \xrightarrow{l} B'_j} \tag{28}$$

Choice over values:

$$\frac{\mathcal{E} \vdash (P \Rightarrow v) \Rightarrow \sigma}{\mathcal{E} \vdash (P \!:=\! \textbf{any } T) \xrightarrow{\delta\sigma} \Pi_{\textbf{stop}}} \tag{29}$$

Generalized parallel:

$$\frac{(\exists \Sigma' \in \Sigma(\mathit{gate}(l)))\ (\forall j \in 1..n) \left(\begin{array}{l}\textbf{if } j \in \Sigma_{i_0} \textbf{ then } Bj = B'_j \\ \textbf{else } \mathcal{E} \vdash B_j \xrightarrow{l} B'_j\end{array}\right)}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{par } \{\, G_i \!:\! \Sigma_i \,\}_{i \in 0..p} \textbf{ in} \\ \quad B_0 \{\ ||\ \ B_k \,\}_{k \in 1..n} \\ \textbf{endpar}\end{array}\right) \xrightarrow{l} \left(\begin{array}{l}\textbf{par } \{\, G_i \!:\! \Sigma_i \,\}_{i \in 0..p} \textbf{ in} \\ \quad B'_0 \{\ ||\ \ B'_k \,\}_{k \in 1..n} \\ \textbf{endpar}\end{array}\right)} \quad [\mathit{gate}(l) \in \{G_i\}] \tag{30}$$

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{l} B'_j}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{par } \{ G_i : \Sigma_i \}_{i \in 0..p} \textbf{ in} \\ \quad B_0 \{ \,||\, B_k \}_{k \in 1..n} \\ \textbf{endpar}\end{array}\right) \xrightarrow{l} \left(\begin{array}{l}\textbf{par } \{ G_i : \Sigma_i \}_{i \in 0..p} \textbf{ in} \\ \quad B_0 \,||\, \ldots B'_j \ldots \,||\, B_n \\ \textbf{endpar}\end{array}\right)} \quad [gate(l) \notin \{G_i\}] \tag{31}$$

Parallel over values:

$$\frac{\begin{array}{l}\mathcal{E} \vdash VL \Rightarrow (v_1, \ldots, v_n) \Rightarrow \sigma \\ \mathcal{E} \vdash B_0[\sigma] \xrightarrow{l} B'_0\end{array}}{\mathcal{E} \vdash \textbf{par } VL \;||\, B_0 \textbf{ endpar} \xrightarrow{l} B'_0} \tag{32}$$

General case:

$$\frac{\begin{array}{l}\mathcal{E} \vdash M_0 \Rightarrow \sigma \\ \mathcal{E} \vdash B_0[\sigma] \xrightarrow{l} B'_0\end{array}}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{case} \\ \quad \{ M_k \rightarrow B_k \}_{k \in 0..p} \\ \textbf{endcase}\end{array}\right) \xrightarrow{l} B'_0} \tag{33}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash M_0 \Rightarrow \textbf{fail} \\ \mathcal{E} \vdash (\textbf{case } \{ M_k \rightarrow B_k \}_{k \in 1..p} \textbf{ endcase}) \xrightarrow{l} B'\end{array}}{\mathcal{E} \vdash (\textbf{case } \{ M_k \rightarrow B_k \}_{k \in 0..p} \textbf{ endcase}) \xrightarrow{l} B'} \tag{34}$$

Local declaration:

$$\frac{\mathcal{E} \vdash B_0 \xrightarrow{l} B'_0}{\mathcal{E} \vdash \textbf{local } VL \textbf{ in } B_0 \textbf{ endloc} \xrightarrow{l} B'_0} \tag{35}$$

Hiding:

$$\frac{\mathcal{E} \vdash B_0 \xrightarrow{l} B'_0}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{hide } \{ G_i : \Gamma_i \}_{i \in 1..n} \textbf{ in} \\ \quad B_0 \\ \textbf{endhide}\end{array}\right) \xrightarrow{l} B'_0} \quad gate(l) \notin \{G_i \mid i \in 1..n\} \tag{36}$$

$$\frac{\mathcal{E} \vdash B_0 \xrightarrow{l} B_0'}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{hide } \{\, G_i \,\}_{i \in 1..n} \textbf{ in} \\ \quad B_0 \\ \textbf{endhide}\end{array}\right) \xrightarrow{\texttt{i}} B_0'} \qquad gate(l) \in \{G_i \mid i \in 1..n\} \tag{37}$$

Process instantiation:

$$\frac{\begin{array}{l}\mathcal{E} \vdash \Pi \Rightarrow \texttt{[}gl\texttt{]} \rightarrow (vl) \rightarrow tl \rightarrow B \\ \mathcal{E} \vdash VL \Rightarrow vl \Rightarrow \sigma \\ \mathcal{E} \vdash \textbf{rename } gl\texttt{:=}GL \textbf{ in } B[\sigma] \xrightarrow{l} B'\end{array}}{\mathcal{E} \vdash \Pi\texttt{[}GL\texttt{]}\texttt{(}VL\texttt{)} \xrightarrow{l} B'} \tag{38}$$

Post renaming:

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{G_j vl} B' \\ \mathcal{E} \vdash VL_j \Rightarrow vl \Rightarrow \sigma\end{array}}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{rename} \\ \quad (G_i \; VL_i\texttt{:=}G_i' vl_i)^* \\ \textbf{in } B \textbf{ endren}\end{array}\right) \xrightarrow{G_j' vl_j[\sigma]} \left(\begin{array}{l}\textbf{rename} \\ \quad (G_i \; VL_i\texttt{:=}G_i' vl_i)^* \\ \textbf{in } B' \textbf{ endren}\end{array}\right)} \tag{39}$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{l} B'}{\mathcal{E} \vdash \left(\begin{array}{l}\textbf{rename} \\ \quad (G_i \; VL_i\texttt{:=}G_i' vl_i)^* \\ \textbf{in } B \textbf{ endren}\end{array}\right) \xrightarrow{l} \left(\begin{array}{l}\textbf{rename} \\ \quad (G_i \; VL_i\texttt{:=}G_i' vl_i)^* \\ \textbf{in } B' \textbf{ endren}\end{array}\right)} \qquad [gate(l) \notin \{G_i\}] \tag{40}$$

5 Conclusion

We have proposed a User Language for E-LOTOS that covers both the data part and the behaviour part of E-LOTOS in a symmetric way. It also includes desirable features such as gate typing.

However, this proposal is by no means complete. It remains to be completed and integrated with the other proposed enhancements.

A Expressing the "if" operator in standard LOTOS

We give here a concrete syntax for the "**if**" operator defined in Section 3.5:

```
<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:

$$\begin{array}{l}\textbf{if } E_0 \textbf{ then } B_0\\ \textbf{elsif } E_1 \textbf{ then } B_1\\ \textbf{elsif } E_2 \textbf{ then } B_2\\ \ldots\\ \textbf{elsif } E_n \textbf{ then } B_n\\ \textbf{else } B_{n+1}\\ \textbf{endif}\end{array}$$

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[6].

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:

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

```
(
 let X0:bool = E0 in
   (
   [X0] -> ([[B0]])
   []
   [not (X0)] ->
     let X1:bool = E1 in
       (
       [X1] -> ([[B1]])
       []
       [not (X1)] ->
         let X2:bool = E2 in
           (
           [X2] -> ([[B2]])
           []
           [not (X2)] ->
             ...
                   let Xn:bool = En in
                     (
                     [Xn] -> ([[Bn]])
                     []
                     [not (Xn)] -> ([[Bn+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] -> ([[B1]])
  []
  [not (X)] -> ([[B2]])
  )
)
```

□

B Sequential composition and bracketed syntax in LOTCAL

In connection with his proposal for a bracketed syntax (see Section 3.9 above), Ed Brinksma proposed a syntactic unification of the operators ";" and ">>" (see Section 3.14).

We give here a synthetic view of his 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]):

$$
\begin{array}{rcl}
B & ::= & NonSeqB \\
 & | & SeqB \\
NonSeqB & ::= & \textbf{stop} \\
 & | & \textbf{exit} \; ... \\
 & | & \textbf{dis} \; B_1 \; \texttt{[>} \; B_2 \; \textbf{enddis} \\
 & | & \textbf{sel} \; B_1 \; \texttt{[]} \; ... \; \texttt{[]} \; B_n \; \textbf{endsel} \\
 & | & \textbf{par} \; ...B_1 \; \texttt{||} \; ... \; \texttt{||} \; B_n \; \textbf{endpar} \\
 & | & \textbf{hide} \; ... \; \textbf{in} \; B \\
 & | & \textbf{let} \; ... \; \textbf{in} \; B \\
 & | & P[...](...) \\
SeqB & ::= & G... \; ; \; B \\
 & | & Block \; [\textbf{init} \; V_1 : T_n, ..., V_n : T_n] \; ; \; B \\
Block & ::= & NonSeqB \\
 & | & \textbf{seq} \; SeqB \; \textbf{endseq}
\end{array}
$$

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

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

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

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

C 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 3.18).

```
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
```

Note: Nested processes have been "flattened" according to the decision mentioned in Section 3.16. E-LOTOS modules should replace the abstraction facilities provided by the "**where**" clause.

```
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 [...]
endproc

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

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

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

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

endspec
```

D 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 3.5, 3.18, and 3.19).

```
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
```

Note: Nested processes have been “flattened” according to the decision mentioned in Section 3.16.

E-LOTOS modules should replace the abstraction facilities provided by the "**where**" clause.

```
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)
endproc

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

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

process RECEIVE [GET, RDT, RACK] (BASE:NUM, 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 [GET, RDT, RACK] (BASE:NUM, 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

endspec
```

E Definition of LOTOS standard data types (IS 8007) in E-LOTOS

This Annex shows how the standard library of Lotos [ISO88b] can be expressed using the proposed User Language for E-Lotos. We follow the following principles for this translation:

- Whenever possible, we keep the existing names used for types, functions and sorts
- The existing algebraic equations are kept and grouped, for each function definition, in an section introduced by the "**eqns**" keyword. However, equations which are not rewrite rules, and equations between constructors have been replaced with rewrite rules.
- As the modules language for E-Lotos is not fixed yet, we have used the only proposal available yet: the so-called "Lotosphere proposal", which is defined in the output document of the Yokohama SC21 meeting, July 1993. In particular, we introduced a separation between interfaces (signatures) and modules. The following table summarizes the keyword translation rules between Lotos types and Lotosphere modules.

Lotos keywords	E-Lotos keywords
type	**module** and **signature**
endtype	**endmod** and **endsig**
sorts	**type** ...**endtype**
opns	**function** ...**endfunc**
sortnames	**types**
opnnames	**fnnames**
renamedby	{...}
actualizedby	[...]

E.1 Boolean

signature Boolean **is**
  type Bool **is**
    true
  | false
  endtype
  function not (x: Bool) : Bool
  function _and_ (x: Bool, y: Bool) : Bool
  function _or_ (x: Bool, y: Bool) : Bool
  function _xor_ (x: Bool, y: Bool) : Bool
  function _implies_ (x: Bool, y: Bool) : Bool
  function _iff_ (x: Bool, y: Bool) : Bool
  function _eq_ (x: Bool, y: Bool) : Bool **reqs equality** (eq)
  function _ne_ (x: Bool, y: Bool) : Bool
endsig Boolean

```
module Boolean is
   type Bool is
      true
   |  false
   endtype
   function not (x: Bool) : Bool is
      case x in
         true -> false
      |  false -> true
      endcase
   eqns
        not(true) = false;
        not(false) = true;
   endfunc
   function _and_ (x: Bool, y: Bool) : Bool is
      case y in
         true -> x
      |  false -> false
      endcase
   eqns forall x: Bool
        x and true = x;
        x and false = false;
   endfunc
   function _or_ (x: Bool, y: Bool) : Bool is
      case y in
         true -> true
      |  false -> x
      endcase
   eqns forall x: Bool
        x or true = true;
        x or false = x;
   endfunc
   function _xor_ (x: Bool, y: Bool) : Bool is x and not(y) or (y and not(x))
   eqns forall x, y: Bool
        x xor y = x and not(y) or (y and not(x));
   endfunc
   function _implies_ (x: Bool, y: Bool) : Bool is y or not(x)
```

eqns forall x, y: Bool

x implies y = y or not(x);

endfunc

function _iff_ (x: Bool, y: Bool) : Bool **is** x implies y and (y implies x)

eqns forall x, y: Bool

x iff y = x implies y and (y implies x);

endfunc

function _eq_ (x: Bool, y: Bool) : Bool **reqs equality** (eq)

eqns forall x, y: Bool

x eq y = x iff y;

function _ne_ (x: Bool, y: Bool) : Bool **is** x xor y

eqns forall x: Bool

x ne y = x xor y;

endfunc

endmod Boolean

E.2 FBoolean

signature FBoolean **is**

type FBool

function true : FBool

function not (x: FBool) : FBool

eqns forall x: FBool

not(not(x)) = x;

endsig FBoolean

E.3 Element

signature Element **is**

FBoolean +

type Element

function _eq_ (x: Element, y: Element) : FBool **reqs equality** (eq)

function _ne_ (x: Element, y: Element) : FBool

eqns forall x, y: Element

x ne y = not(x eq y);

endsig Element

E.4 Set

```
signature Set is
   formal Element
   import Boolean + NaturalNumber
   type Set
   function _eq_ (s: Set, t: Set) : Bool reqs equality (eq)
   function _ne_ (s: Set, t: Set) : Bool
   function {} : Set
   function Insert (x: Element, s: Set) : Set
   function Remove (x: Element, s: Set) : Set
   function _IsIn_ (x: Element, s: Set) : Bool
   function _NotIn_ (x: Element, s: Set) : Bool
   function _Union_ (s: Set, t: Set) : Set
   function _Ints_ (s: Set, t: Set) : Set
   function _Minus_ (s: Set, t: Set) : Set
   function _Includes_ (s: Set, t: Set) : Bool
   function _IsSubsetOf_ (s: Set, t: Set) : Bool
   function Card (s: Set) : Nat
endsig Set
module Set is
   formal Element
   import Boolean + NaturalNumber
   type Set is
      {}
   |  Add (e: Element, s: Set)
   endtype
   function _eq_ (s: Set, t: Set) : Bool
   eqns  s eq t = s Includes t and (t Includes s);
   endfunc
   function _ne_ (s: Set, t: Set) : Bool is not(s eq t)
   eqns  forall s, t: Set
      s ne t = not(s eq t)
   endfunc
   function Insert (x: Element, s: Set) : Set is
      case s in
        {} -> Add (x, {})
```

```
    | Add (e:= y: Element, s:= t: Set) when x eq y -> Add (x, t)
    | Add (e:= y: Element, s:= t: Set) when x ne y -> Add (y, Insert (x, t))
    endcase
eqns forall x, y: Element, s: Set
    Insert (x, {}) = Add (x, {});
    x eq y => Insert (x, Add (y, s)) = Add (x, s);
    x ne y => Insert(x, Add (y, s)) = Add (y, Insert (x, s));
endfunc
function Remove (x: Element, s: Set) : Set is
    case s in
      {} -> {}
    | Add (e:= y: Element, s:= t: Set) when x eq y -> Remove (x, t)
    | Add (e:= y: Element, s:= t: Set) when x ne y -> Add (y, Remove (x, t))
    endcase
eqns forall x, y: Element, s: Set
    Remove (x, {}) = {};
    x eq y => Remove (x, Add (y, s)) = Remove (x, s);
    x ne y => Remove (x, Add (y, s)) = Add (y, Remove (x, s));
endfunc
function _IsIn_ (x: Element, s: Set) : Bool is
    case s in
      {} -> false
    | Add (e:= y: Element, s:= t: Set) when x eq y -> true
    | Add (e:= y: Element, s:= t: Set) when x ne y -> x IsIn t
    endcase
eqns forall x, y: Element, s: Set
    x IsIn {} = false;
    x eq y => x IsIn Add (y, s) = true;
    x ne y => x IsIn Add (y, s) = x IsIn s;
endfunc
function _NotIn_ (e: Element, s: Set) : Bool is not(x IsIn s)
eqns forall x: Element, s: Set
    x NotIn s = not (x IsIn s);
endfunc
function _Union_ (s: Set, t: Set) : Set is
    case s in
      {} -> t
```

```
   | Add (e:= x: Element, s:= s1: Set) -> Add (x, s1 Union t)
   endcase
eqns forall x: Element, s, t: Set
   {} Union s = s;
   Add (x, s) Union t = Add (x, s Union t);
endfunc
function _Ints_ (s: Set, t: Set) : Set is
   case s in
     {} -> {}
   | Add (e:= x: Element, s:= s1: Set) when x IsIn t -> Add (x, s1 Ints t)
   | Add (e:= y: Element, s:= s1: Set) when x NotIn t -> s1 Ints t
   endcase
eqns forall x: Element, s, t: Set
   {} Ints s = {};
   x IsIn t => Add (x, s) Ints t = Add (x, s Ints t);
   x NotIn t => Add (x, s) Ints t = s Ints t;
endfunc
function _Minus_ (s: Set, t: Set) : Set is
   case t in
     {} -> s
   | Add (e:= x: Element, s:= t1: Set) -> Remove (x, s) Minus t1
   endcase
eqns forall x: Element, s, t: Set
   s Minus {} = s;
   s Minus Add (x, t) = Remove (x, s) Minus t
endfunc
function _Includes_ (s: Set, t: Set) : Bool is
   case t in
     {} -> true
   | Add (e:= x: Element, s:= t1: Set) -> x IsIn s and (s Includes t1)
   endcase
eqns forall x: Element, s, t: Set
   s Includes {} = true;
   s Includes Add (x, t) = x IsIn s and (s Includes t);
endfunc
function _IsSubsetOf_ (s: Set, t: Set) : Bool is t Includes s
eqns forall s, t: Set
```

```
        s IsSubsetOf t = t Includes s;
    endfunc
    function Card (s: Set) : Nat is
        case s in
          {} -> 0
        | Add (s:= s1: Set, ...) -> Succ (Card (s1))
        endcase
    eqns forall x: Element, s: Set
        Card ({}) = 0;
        Card (Add (x, s)) = Succ (Card (s))
    endfunc
endmod Set
```

E.5 BasicNonEmptyString

```
signature BasicNonEmptyString is
    formal Element
    type NonEmptyString is
        String (e: Element)
    |   _+_ (e: Element, nes: NonEmptyString)
    endtype
endsig BasicNonEmptyString
module BasicNonEmptyString is
    formal Element
    type NonEmptyString is
        String (e: Element)
    |   _+_ (e: Element, nes: NonEmptyString)
    endtype
endmod BasicNonEmptyString
```

E.6 RicherNonEmptyString

```
signature RicherNonEmptyString is
    import BasicNonEmptyString + NaturalNumber
    function _+_ (s: NonEmptyString, x: Element) : NonEmptyString
    function _++_ (s: NonEmptyString, t: NonEmptyString) : NonEmptyString
    function Reverse (s: NonEmptyString) : NonEmptyString
    function Length (s: NonEmptyString) : Nat
```

```
endsig RicherNonEmptyString
module RicherNonEmptyString is
   import BasicNonEmptyString + NaturalNumber
   function _+_ (s: NonEmptyString, x: Element) : NonEmptyString is
      case s in
        String(e:= y: Element) -> y + String(x)
      | (e:= y: Element) + (nes:= t: NonEmptyString) -> y + (t + x)
      endcase
   eqns forall x, y: Element, s: NonEmptyString
      String(x) + y = x + String(y);
        x + s + y = x + (s + y);
   endfunc
   function _++_ (s: NonEmptyString, t: NonEmptyString) : NonEmptyString is
      case s in
        String(e:= x: Element) -> x + t
      | (e:= x: Element) + (nes:= s1: NonEmptyString) -> x + (s1 ++ t)
      endcase
   eqns forall x: Element, s, t: NonEmptyString
      String(x) ++ s = x + s;
        x + s ++ t = x + (s ++ t);
   endfunc
   function Reverse (s: NonEmptyString) : NonEmptyString is
      case s in
        String(e:= x: Element) -> String (x)
      | (e:= x: Element) + (nes:= s1: NonEmptyString) -> Reverse (s1) + x
      endcase
   eqns forall x: Element, s: NonEmptyString
      Reverse(String(x)) = String(x);
        Reverse(x + s) = Reverse(s) + x;
   endfunc
   function Length (s: NonEmptyString) : Nat is
      case s in
        String(e:= x: Element) -> Succ (0)
      | (e:= x: Element) + (nes:= s1: NonEmptyString) -> Succ (Length (s1))
      endcase
   eqns  forall x: Element, s: NonEmptyString
        Length(String(x)) = Succ(0);
```

```
      Length(x + s) = Succ(Length(s))
   endfunc
endmod RicherNonEmptyString
```

E.7 NonEmptyString

```
signature NonEmptyString is
   import RicherNonEmptyString + Boolean
   function _eq_ (s: NonEmptyString, t: NonEmptyString) : Bool reqs equality (eq)
   function _ne_ (s: NonEmptyString, t: NonEmptyString) : Bool
endsig NonEmptyString
module NonEmptyString is
   import RicherNonEmptyString + Boolean
   function _eq_ (s: NonEmptyString, t: NonEmptyString) : Bool is
      case
        s :: String (e:=x: Element) and t :: String (e:=y: Element)
             when x eq y -> true
      | s :: String (e:=x: Element) and t :: String (e:=y: Element)
             when x ne y -> false
      | (s :: String (...) and t :: ... + ...) or
        (s :: ... + ... and t :: String (...)) -> false
      | s :: (e:= x: Element) + (nes:= s1: NonEmptyString) and
        t :: (e:= y: Element) + (nes:= t1: NonEmptyString)
             when x eq y -> s1 eq t1
      | s :: (e:= x: Element) + (nes:= s1: NonEmptyString) and
        t :: (e:= y: Element) + (nes:= t1: NonEmptyString)
             when x ne y -> false
      endcase
   eqns forall s, t : NonEmptyString, x, y : Element
        x eq y => String(x) eq String(y) = true;
        x ne y => String(x) eq String(y) = false;
        String(x) eq (y + s) = false;
        x + s eq String(y) = false;
        x eq y => x + s eq (y + t) = s eq t;
        x ne y => x + s eq (y + t) = false;
   endfunc
   function _ne_ (s: NonEmptyString, t: NonEmptyString) : Bool is not(s eq t)
```

```
   eqns forall s, t : NonEmptyString
      s ne t = not(s eq t)
   endfunc
endmod NonEmptyString
```

E.8 String

```
signature String
   formal Element
   import NaturalNumber + Boolean
   type String is
      <>
   |  _+_ (e: Element, s: String)
   endtype
   function String (x: Element) : String
   function _+_ (s: String, x: Element) : String
   function _++_ (s: String, t: String) : String
   function Reverse (s: String) : String
   function Length (s: String) : Nat
   function _eq_ (s: String, t: String) : Bool reqs equality (eq)
   function _ne_ (s: NonEmptyString, t: NonEmptyString) : Bool
endsig String
module String
   formal Element
   import NaturalNumber + Boolean
   type String is
      <>
   |  _+_ (e: Element, s: String)
   endtype
   function String (x: Element) : String is x + <>
   eqns forall x: Element
      String (x) = x + <>
   endfunc
   function _+_ (s: String, x: Element) : String is
      case s in
        <> -> x + <>
      | (e:= y: Element) + (s:= s1: String) -> y + (s1 + x)
```

```
    endcase
eqns forall x, y: Element, s: String
    <> + y = x + <>;
      x + s + y = x + (s + y);
endfunc
function _++_ (s: String, t: String) : String is
    case s in
      <> -> t
    | (e:= x: Element) + (s:= s1: String) -> x + (s1 ++ t)
    endcase
eqns forall x: Element, s, t: String
    <> ++ s = s;
      x + s ++ t = x + (s ++ t);
endfunc
function Reverse (s: String) : String is
    case s in
      <> -> <>
    | (e:= x: Element) + (s:= s1: String) -> Reverse (s1) + x
    endcase
eqns forall x: Element, s: String
    Reverse(<>) = <>;
      Reverse(x + s) = Reverse(s) + x;
endfunc
function Length (s: String) : Nat is
    case s in
      <> -> 0
    | (e:= x: Element) + (s:= s1: String) -> Succ (Length (s1))
    endcase
eqns forall x: Element, s: String
      Length(<>) = 0;
      Length(x + s) = Succ(Length(s))
endfunc
function _eq_ (s: String, t: String) : Bool is
    case
      s :: <> and t :: <> -> true
    | (s :: <> and t :: ... + ...) or
      (s :: ... + ... and t :: <>) -> false
```

```
    | s :: (e:= x: Element) + (s:= s1: String) and
      t :: (e:= y: Element) + (s:= t1: String) when x eq y -> s1 eq t1
    | s :: (e:= x: Element) + (s:= s1: String) and
      t :: (e:= y: Element) + (s:= t1: String) when x ne y -> false
    endcase
  eqns forall s, t : String, x, y : Element
      <> eq <> = true;
      <> eq (x + s) = false;
      x + s eq <> = false;
      x eq y => x + s eq (y + t) = s eq t;
      x ne y => x + s eq (y + t) = false;
  endfunc
  function _ne_ (s: String, t: String) : Bool is not(s eq t)
  eqns forall s, t : String
      s ne t = not(s eq t)
  endfunc
endmod String
```

E.9 BasicNaturalNumber

```
signature BasicNaturalNumber is
  type Nat is
    0
  | Succ (N: Nat)
  endtype
  function _+_ (m: Nat, n: Nat) : Nat
  function _*_ (m: Nat, n: Nat) : Nat
  function _**_ (m: Nat, n: Nat) : Nat
endsig BasicNaturalNumber
module BasicNaturalNumber is
  type Nat is
    0
  | Succ (N: Nat)
  endtype
  function _+_ (m: Nat, n: Nat) : Nat is
    case n in
      0 -> m
```

```
      | Succ (N:=n1 : Nat) -> Succ (m) + n
      endcase
   eqns forall  m, n : Nat
        m + 0 = m;
        m + Succ(n) = Succ(m) + n;
   endfunc
   function _*_ (m: Nat, n: Nat) : Nat is
      case n in
        0 -> 0
      | Succ (N:=n1 : Nat) -> m + (m * n1)
      endcase
   eqns forall  m, n : Nat
        m * 0 = 0;
        m * Succ(n) = m + (m * n);
   endfunc
   function _**_ (m: Nat, n: Nat) : Nat is
      case n in
        0 -> 0
      | Succ (N:=n1 : Nat) -> m * (m ** n1)
      endcase
   eqns forall  m, n : Nat
        m ** 0 = Succ(0);
        m ** Succ(n) = m * (m ** n)
   endfunc
endmod BasicNaturalNumber
```

E.10 NaturalNumber

```
signature NaturalNumber is
   import BasicNaturalNumber + Boolean
   function _eq_ (m: Nat, n: Nat) : Bool reqs equality (eq)
   function _ne_ (m: Nat, n: Nat) : Bool
   function _lt_ (m: Nat, n: Nat) : Bool
   function _le_ (m: Nat, n: Nat) : Bool
   function _ge_ (m: Nat, n: Nat) : Bool
   function _gt_ (m: Nat, n: Nat) : Bool
endsig NaturalNumber
```

```
module NaturalNumber is
  import BasicNaturalNumber + Boolean
  function _eq_ (m: Nat, n: Nat) : Bool is
    case
       m :: 0 and n :: 0 -> true
    | (m :: 0 and n :: Succ (...)) or
      (m :: Succ (...) and n :: 0) -> false
    |  m :: Succ (N:=m1: Nat) and n :: Succ (N:=n1: Nat) -> m1 eq n1
    endcase
  eqns forall m, n : Nat
      0 eq 0 = true;
      0 eq Succ(m) = false;
      Succ(m) eq 0 = false;
      Succ(m) eq Succ(n) = m eq n;
  endfunc
  function _ne_ (m: Nat, n: Nat) : Bool is not (m eq n)
  eqns forall m, n : Nat
      m ne n = not(m eq n);
  endfunc
  function _lt_ (m: Nat, n: Nat) : Bool is
    case
      (m :: 0 or m :: Succ (...)) and n :: 0 -> false
    |  m :: 0 and n :: Succ (...) -> true
    |  m :: Succ (N:=m1: Nat) and n :: Succ (N:=n1: Nat) -> m1 lt n1
    endcase
  eqns forall m, n : Nat
      0 lt 0 = false;
      0 lt Succ(n) = true;
      Succ(n) lt 0 = false;
      Succ(m) lt Succ(n) = m lt n;
  endfunc
  function _le_ (m: Nat, n: Nat) : Bool is m lt n or (m eq n)
  eqns forall m, n : Nat
      m le n = m lt n or (m eq n);
  endfunc
  function _ge_ (m: Nat, n: Nat) : Bool is not(m lt n)
  eqns forall m, n : Nat
```

```
      m ge n = not(m lt n);
   endfunc
   function _gt_ (m: Nat, n: Nat) : Bool is not(m le n)
   eqns forall m, n : Nat
      m gt n = not(m le n);
   endfunc
endmod NaturalNumber
```

E.11 NatRepresentations

```
signature NatRepresentations is
   import HexNatRepr + DecNatRepr + OctNatRepr + BitNatRepr
endsig
```

E.12 HexNatRepr

```
signature HexNatRepr is
   import HexString
   function NatNum (hs: HexString) : Nat
endsig
module HexNatRepr is
   import HexString
   function NatNum (hs: HexString) : Nat is
      case hs in
        Hex (h:=h1: HexDigit) -> NatNum (h1)
      | (h:=h1: HexDigit) + (hs:=hs1: HexString) ->
           NatNum(h1) * (Succ(NatNum(F)) ** Length(hs1)) + NatNum(hs1)
      endcase
   eqns forall hs : HexString , h : HexDigit
   NatNum(Hex(h)) = NatNum (h);
   NatNum(h + hs) = NatNum(h) * (Succ(NatNum(F)) ** Length(hs)) + NatNum(hs)
endmod
```

E.13 HexString

```
signature HexString is
   NonEmptyString [HexDigit :: types HexDigit   for Element
                                   Bool        for FBool
                                   HexString  for NonEmptyString
```

```
                    constructors Hex for String
                    labels h  for e
                         hs for s
          ]
endsig
module HexString is
  NonEmptyString [HexDigit :: types HexDigit   for Element
                         Bool       for FBool
                         HexString  for NonEmptyString
                    constructors Hex for String
                    labels h  for e
                         hs for s
          ]
endmod
```

E.14 HexDigit

```
signature HexDigit is
  import Boolean + NaturalNumber
  type HexDigit is
    0| 1| 2| 3| 4| 5| 6| 7| 8| 9| A| B| C| D| E| F
  endtype
  function _eq_ (x: HexDigit, y: HexDigit) : Bool reqs equality (eq)
  function _ne_ (x: HexDigit, y: HexDigit) : Bool
  function _lt_ (x: HexDigit, y: HexDigit) : Bool
  function _le_ (x: HexDigit, y: HexDigit) : Bool
  function _ge_ (x: HexDigit, y: HexDigit) : Bool
  function _gt_ (x: HexDigit, y: HexDigit) : Bool
  function NatNum (x: HexDigit) : Nat
endsig HexDigit
module HexDigit is
  import Boolean + NaturalNumber
  type HexDigit is
    0| 1| 2| 3| 4| 5| 6| 7| 8| 9| A| B| C| D| E| F
  endtype
  function _eq_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) eq NatNum(y)
  eqns forall x, y : HexDigit
```

```
    x eq y = NatNum(x) eq NatNum(y);
endfunc
function _ne_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) ne NatNum(y)
eqns forall x, y : HexDigit
    x ne y = NatNum(x) ne NatNum(y);
endfunc
function _lt_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) lt NatNum(y)
eqns forall x, y : HexDigit
    x lt y = NatNum(x) lt NatNum(y);
endfunc
function _le_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) le NatNum(y)
eqns forall x, y : HexDigit
    x le y = NatNum(x) le NatNum(y);
endfunc
function _ge_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) ge NatNum(y)
eqns forall x, y : HexDigit
    x ge y = NatNum(x) ge NatNum(y);
endfunc
function _gt_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) gt NatNum(y)
eqns forall x, y : HexDigit
    x gt y = NatNum(x) gt NatNum(y);
endfunc
function NatNum (x: HexDigit) : Nat is
  case x in
    0 -> 0
  | 1 -> Succ(NatNum(0))
  | 2 -> Succ(NatNum(1))
  | 3 -> Succ(NatNum(2))
  | 4 -> Succ(NatNum(3))
  | 5 -> Succ(NatNum(4))
  | 6 -> Succ(NatNum(5))
  | 7 -> Succ(NatNum(6))
  | 8 -> Succ(NatNum(7))
  | 9 -> Succ(NatNum(8))
  | A -> Succ(NatNum(9))
  | B -> Succ(NatNum(A))
  | C -> Succ(NatNum(B))
```

| D -> Succ(NatNum(C))
| E -> Succ(NatNum(D))
| F -> Succ(NatNum(E))
endcase
eqns
NatNum(0) = 0;
NatNum(1) = Succ(NatNum(0));
NatNum(2) = Succ(NatNum(1));
NatNum(3) = Succ(NatNum(2));
NatNum(4) = Succ(NatNum(3));
NatNum(5) = Succ(NatNum(4));
NatNum(6) = Succ(NatNum(5));
NatNum(7) = Succ(NatNum(6));
NatNum(8) = Succ(NatNum(7));
NatNum(9) = Succ(NatNum(8));
NatNum(A) = Succ(NatNum(9));
NatNum(B) = Succ(NatNum(A));
NatNum(C) = Succ(NatNum(B));
NatNum(D) = Succ(NatNum(C));
NatNum(E) = Succ(NatNum(D));
NatNum(F) = Succ(NatNum(E))
endfunc
endmod HexDigit

E.15 DecNatRepr

signature DecNatRepr **is**
import NaturalNumber + DecString
function NatNum (ds: DecString) : Nat
endsig
module DecNatRepr **is**
import NaturalNumber + DecString
function NatNum (ds: DecString) : Nat **is**
case ds **in**
Dec (d:=d1: DecString) -> NatNum (d1)
| (d:=d1: DecDigit) + (ds:=ds1: DecString) ->
NatNum(d1) * (Succ(NatNum(9)) ** Length(ds1)) + NatNum(ds1)

```
      endcase
   eqns forall ds : DecString , d : DecDigit
   NatNum(Dec(d)) = NatNum(d);
   NatNum(d + ds) = NatNum(d) * (Succ(NatNum(9)) ** Length(ds)) + NatNum(ds)
   endfunc
endmod
```

E.16 DecString

```
signature DecString is
   NonEmptyString [ DecDigit :: types DecDigit   for Element
                                      Bool        for FBool
                                      DecString  for NonEmptyString
                              constructors Dec for String
                              labels d  for e
                                      ds for s
                 ]
endsig
module DecString is
   NonEmptyString [ DecDigit :: types DecDigit   for Element
                                      Bool        for FBool
                                      DecString  for NonEmptyString
                              constructors Dec for String
                              labels d  for e
                                      ds for s
                 ]
endmod
```

E.17 DecDigit

```
signature DecDigit is
   import NaturalNumber + Boolean
   type DecDigit is
      0| 1| 2| 3| 4| 5| 6| 7| 8| 9
   endtype
   function _eq_ (x: DecDigit, y: DecDigit) : Bool reqs equality (eq)
   function _ne_ (x: DecDigit, y: DecDigit) : Bool
   function _lt_ (x: DecDigit, y: DecDigit) : Bool
```

```
   function _le_ (x: DecDigit, y: DecDigit) : Bool
   function _ge_ (x: DecDigit, y: DecDigit) : Bool
   function _gt_ (x: DecDigit, y: DecDigit) : Bool
   function NatNum  (x: DecDigit) : Nat
endsig
module DecDigit is
   import NaturalNumber + Boolean
   type DecDigit is
      0| 1| 2| 3| 4| 5| 6| 7| 8| 9
   endtype
   function _eq_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) eq NatNum(y)
   eqns forall x, y : DecDigit
      x eq y = NatNum(x) eq NatNum(y);
   endfunc
   function _ne_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) ne NatNum(y)
   eqns forall x, y : DecDigit
      x ne y = NatNum(x) ne NatNum(y);
   endfunc
   function _lt_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) lt NatNum(y)
   eqns forall x, y : DecDigit
      x lt y = NatNum(x) lt NatNum(y);
   endfunc
   function _le_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) le NatNum(y)
   eqns forall x, y : DecDigit
      x le y = NatNum(x) le NatNum(y);
   endfunc
   function _ge_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) ge NatNum(y)
   eqns forall x, y : DecDigit
      x ge y = NatNum(x) ge NatNum(y);
   endfunc
   function _gt_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) gt NatNum(y)
   eqns forall x, y : DecDigit
      x gt y = NatNum(x) gt NatNum(y);
   endfunc
   function NatNum  (x: DecDigit) : Nat is
     case x in
      0 -> 0
```

```
      | 1 -> Succ(NatNum(0))
      | 2 -> Succ(NatNum(1))
      | 3 -> Succ(NatNum(2))
      | 4 -> Succ(NatNum(3))
      | 5 -> Succ(NatNum(4))
      | 6 -> Succ(NatNum(5))
      | 7 -> Succ(NatNum(6))
      | 8 -> Succ(NatNum(7))
      | 9 -> Succ(NatNum(8))
      endcase
   eqns forall x, y : DecDigit
        NatNum(0) = 0;
        NatNum(1) = Succ(NatNum(0));
        NatNum(2) = Succ(NatNum(1));
        NatNum(3) = Succ(NatNum(2));
        NatNum(4) = Succ(NatNum(3));
        NatNum(5) = Succ(NatNum(4));
        NatNum(6) = Succ(NatNum(5));
        NatNum(7) = Succ(NatNum(6));
        NatNum(8) = Succ(NatNum(7));
        NatNum(9) = Succ(NatNum(8));
   endfunc
endmod
```

E.18 OctNatRepr

```
signature OctNatRepr is
   import OctString
   function NatNum (os: OctString) : Nat
endsig
module OctNatRepr is
   import OctString
   function NatNum (os: OctString) : Nat is
      case os in
        Oct(o:=o1: OctDigit) -> NatNum(o1)
      | (o:=o1: OctDigit) + (os:=os1: OctString) ->
             NatNum(o1) * (Succ(NatNum(7)) ** Length(os1)) + NatNum(os1)
```

```
      endcase
   eqns forall os : OctString, o : OctDigit
   NatNum(Oct(o)) = NatNum(o);
   NatNum(o + os) = NatNum(o) * (Succ(NatNum(7)) ** Length(os)) + NatNum(os)
   endfunc
endmod
```

E.19 OctString

```
signature OctString is
   NonEmptyString [OctDigit :: types OctDigit  for Element
                                 Bool       for FBool
                                 OctString for NonEmptyString
                          constructors Oct  for String
                          labels o  for e
                                 os for s
                  ]
endsig
module OctString is
   NonEmptyString [OctDigit :: types OctDigit  for Element
                                 Bool       for FBool
                                 OctString for NonEmptyString
                          constructors Oct for String
                          labels o  for e
                                 os for s
                  ]
endmod
```

E.20 OctDigit

```
signature OctDigit is
   import NaturalNumber + Boolean
   type OctDigit is
      0| 1| 2| 3| 4| 5| 6| 7
   endtype
   function _eq_ (x: OctDigit, y: OctDigit) : Bool reqs equality (eq)
   function _ne_ (x: OctDigit, y: OctDigit) : Bool
   function _lt_ (x: OctDigit, y: OctDigit) : Bool
```

```
   function _le_ (x: OctDigit, y: OctDigit) : Bool
   function _ge_ (x: OctDigit, y: OctDigit) : Bool
   function _gt_ (x: OctDigit, y: OctDigit) : Bool
   function NatNum (x: OctDigit) : Nat
endsig OctDigit
module OctDigit is
   import NaturalNumber + Boolean
   type OctDigit is
      0| 1| 2| 3| 4| 5| 6| 7
   endtype
   function _eq_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) eq NatNum(y)
   eqns forall x, y : OctDigit
        x eq y = NatNum(x) eq NatNum(y);
   endfunc
   function _ne_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) ne NatNum(y)
   eqns forall x, y : OctDigit
        x ne y = NatNum(x) ne NatNum(y);
   endfunc
   function _lt_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) lt NatNum(y)
   eqns forall x, y : OctDigit
        x lt y = NatNum(x) lt NatNum(y);
   endfunc
   function _le_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) le NatNum(y)
   eqns forall x, y : OctDigit
        x le y = NatNum(x) le NatNum(y);
   endfunc
   function _ge_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) ge NatNum(y)
   eqns forall x, y : OctDigit
        x ge y = NatNum(x) ge NatNum(y);
   endfunc
   function _gt_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) gt NatNum(y)
   eqns forall x, y : OctDigit
        x gt y = NatNum(x) gt NatNum(y)
   endfunc
   function NatNum (x: OctDigit) : Nat is
      case x in
        0 -> 0
```

```
    | 1 -> Succ(NatNum(0))
    | 2 -> Succ(NatNum(1))
    | 3 -> Succ(NatNum(2))
    | 4 -> Succ(NatNum(3))
    | 5 -> Succ(NatNum(4))
    | 6 -> Succ(NatNum(5))
    | 7 -> Succ(NatNum(6))
    endcase
  eqns forall x: OctDigit
     NatNum(0) = 0;
     NatNum(1) = Succ(NatNum(0));
     NatNum(2) = Succ(NatNum(1));
     NatNum(3) = Succ(NatNum(2));
     NatNum(4) = Succ(NatNum(3));
     NatNum(5) = Succ(NatNum(4));
     NatNum(6) = Succ(NatNum(5));
     NatNum(7) = Succ(NatNum(6))
  endfunc
endmod OctDigit
```

E.21 BitNatRepr

```
signature BitNatRepr is
  import BitString
  function NatNum (bs: BitString) : Nat
endsig
module BitNatREpr is
  import BitString
  function NatNum (bs: BitString) : Nat is
    case bs in
     Bit (b:=b1: Bit) -> NatNum (b)
    | (b:=b1: Bit) + (bs:=bs1: BitString) ->
        NatNum(b1) * (Succ(NatNum(1)) ** Length(bs1)) + NatNum(bs1)
    endcase
  eqns forall  bs : BitString, b : Bit
  NatNum(Bit(b)) = NatNum(b);
  NatNum(b + bs) = NatNum(b) * (Succ(NatNum(1)) ** Length(bs)) + NatNum(bs)
```

```
   endfunc
endmod
```

E.22 BitString

```
signature BitString is
   NonEmptyString [ Bit :: types Bit         for Element
                             Bool       for FBool
                             BitString  for NonEmptyString
                       constructors Bit for String
                       labels b  for e
                              bs for s
                 ]
endsig
module BitString is
   NonEmptyString [ Bit :: types Bit         for Element
                             Bool       for FBool
                             BitString  for NonEmptyString
                       constructors Bit for String
                       labels b  for e
                              bs for s
                 ]
endmod
```

E.23 Bit

```
signature Bit is
   import NaturalNumber + Boolean
   type Bit is
      0| 1
   endtype
   function _eq_ (x: Bit, y: Bit) : Bool reqs equality (eq)
   function _ne_ (x: Bit, y: Bit) : Bool
   function _lt_ (x: Bit, y: Bit) : Bool
   function _le_ (x: Bit, y: Bit) : Bool
   function _ge_ (x: Bit, y: Bit) : Bool
   function _gt_ (x: Bit, y: Bit) : Bool
   function NatNum (x: Bit) : Nat
```

```
endsig Bit
module Bit is
   import NaturalNumber + Boolean
   type Bit is
      0| 1| 2| 3| 4| 5| 6| 7
   endtype
   function _eq_ (x: Bit, y: Bit) : Bool is NatNum(x) eq NatNum(y)
   eqns forall x, y : Bit
       x eq y = NatNum(x) eq NatNum(y);
   endfunc
   function _ne_ (x: Bit, y: Bit) : Bool is NatNum(x) ne NatNum(y)
   eqns forall x, y : Bit
       x ne y = NatNum(x) ne NatNum(y);
   endfunc
   function _lt_ (x: Bit, y: Bit) : Bool is NatNum(x) lt NatNum(y)
   eqns forall x, y : Bit
       x lt y = NatNum(x) lt NatNum(y);
   endfunc
   function _le_ (x: Bit, y: Bit) : Bool is NatNum(x) le NatNum(y)
   eqns forall x, y : Bit
       x le y = NatNum(x) le NatNum(y);
   endfunc
   function _ge_ (x: Bit, y: Bit) : Bool is NatNum(x) ge NatNum(y)
   eqns forall x, y : Bit
       x ge y = NatNum(x) ge NatNum(y);
   endfunc
   function _gt_ (x: Bit, y: Bit) : Bool is NatNum(x) gt NatNum(y)
   eqns forall x, y : Bit
       x gt y = NatNum(x) gt NatNum(y)
   endfunc
   function NatNum (x: Bit) : Nat is
      case x in
       0 -> 0
      | 1 -> Succ(NatNum(0))
      endcase
   eqns forall x: Bit
       NatNum(0) = 0;
```

```
    NatNum(1) = Succ(NatNum(0));
  endfunc
endmod Bit
```

E.24 Octet

```
signature Octet is
  import Bit + Boolean
  type Octet is
    Octet (b1: Bit, b2: Bit, b3: Bit, b4: Bit, b5: Bit, b6: Bit, b7: Bit, b8:Bit)
  endtype
  function Bit1 (o: Octet) : Bit
  function Bit2 (o: Octet) : Bit
  function Bit3 (o: Octet) : Bit
  function Bit4 (o: Octet) : Bit
  function Bit5 (o: Octet) : Bit
  function Bit6 (o: Octet) : Bit
  function Bit7 (o: Octet) : Bit
  function Bit8 (o: Octet) : Bit
  function _eq_ (x: Octet, y: Octet) : Bool reqs equality (eq)
  function _ne_ (x: Octet, y: Octet) : Bool
endsig
module Octet is
  import Bit + Boolean
  type Octet is
    Octet (b1: Bit, b2: Bit, b3: Bit, b4: Bit, b5: Bit, b6: Bit, b7: Bit, b8:Bit)
  endtype
  function Bit1 (o: Octet) : Bit is o.b1
  eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
     Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b1;
  endfunc
  function Bit2 (o: Octet) : Bit is o.b2
  eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
     Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b2;
  endfunc
  function Bit3 (o: Octet) : Bit is o.b3
  eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
```

Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b3;

endfunc

function Bit4 (o: Octet) : Bit **is** o.b4

eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit

Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b4;

endfunc

function Bit5 (o: Octet) : Bit **is** o.b5

eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit

Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b5;

endfunc

function Bit6 (o: Octet) : Bit **is** o.b6

eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit

Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b6;

endfunc

function Bit7 (o: Octet) : Bit **is** o.b7

eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit

Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b7;

endfunc

function Bit8 (o: Octet) : Bit **is** o.b8

eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit

Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b8;

endfunc

function _eq_ (x: Octet, y: Octet) : Bool

eqns forall x, y : Octet

x eq y = Bit1(x) eq Bit1(y) and (Bit2(x) eq Bit2(y)) and
(Bit3(x) eq Bit3(y)) and (Bit4(x) eq Bit4(y)) and
(Bit5(x) eq Bit5(y)) and (Bit6(x) eq Bit6(y)) and
(Bit7(x) eq Bit7(y)) and (Bit8(x) eq Bit8(y));

endfunc

function _ne_ (x: Octet, y: Octet) : Bool **is** not(x eq y)

eqns forall x, y : Octet

x ne y = not(x eq y)

endfunc

endmod Octet

E.25 OctetString

```
signature OctetString is
  String [ Octet :: types Octet       for Element
                          Bool        for FBool
                          OctetString for String
                    funnames Octet for String
         ]
endsig
module OctetString is
  String [ Octet :: types Octet       for Element
                          Bool        for FBool
                          OctetString for String
                    labels o  for e
                           os for s
                    funnames Octet for String
         ]
endmod
```

F Translation of LOTOSPHERE data types in E-LOTOS

This Annex indicates how some of the data types of the so-called "LOTOSPHERE proposal" (output document of the Yokohama SC21 meeting, July 1993) can be expressed using the proposed User Language for E-LOTOS. It is worth noticing that:

- The LOTOSPHERE proposal only defines interfaces (signatures) rather than the implementation of modules themselves.
- In this Annex, we stress the differences (i.e., enhancements) between LOTOS standard library and the LOTOSPHERE proposal. We do not repeat thoses definitions which are similar as those given in Annex E.
- The LOTOSPHERE proposal uses "`==`", "`!=`", "`<`", "`<=`", "`>`", and "`>=`" where the LOTOS library uses "`eq`", "`ne`", "`lt`", "`le`", "`gt`", and "`ge`". In E-LOTOS, we might wish to allow both notation to ensure both upward compatibility and user-friendliness.

F.1 BOOL

Same as in Annex E, except that "`implies`" and "`iff`" are respectively noted "`==>`" and "`<==>`".

F.2 NAT

In the LOTOSPHERE proposal, natural numbers are a built-in type, not defined by "`0`" and "`Succ`" constructors. Natural numbers have an arbitrary, unlimited precision. They can be written using decimal notation (e.g., "`123`"), but also in binary, octal, or hexadecimal notations (e.g., "`bin (100101101)`", "`oct (455)`", "`hex (12d)`".

A unary function "`pred`" and a substraction operator are available, which raise an underflow error if a negative number is to be returned. Similarly, division and modulo operators are available, which can raise a division-by-zero error.

signature NAT **is**

  type Nat **is** <decimal digit sequence> **endtype**

  function bin (<binary digit sequence>) : Nat

  function oct (<octal digit sequence>) : Nat

  function hex (<hexadecimal digit sequence>) : Nat

  function succ (Nat) : Nat

  function pred [UNDERFLOW] : Nat

  function _+_ (Nat, Nat) : Nat

  function _-_ [UNDERFLOW] (Nat, Nat) : Nat

  function _*_ (Nat, Nat) : Nat

  function _/_ [ZERO_DVISION] (Nat, Nat) : Nat

  function rem [ZERO_DVISION] (Nat, Nat) : Nat

  function pow (Nat, Nat) : Nat

  function max (Nat, Nat) : Nat

function min (Nat, Nat) : Nat
function _>_ (Nat, Nat) : Bool
function _>=_ (Nat, Nat) : Bool
function _<_ (Nat, Nat) : Bool
function _<=_ (Nat, Nat) : Bool
function _==_ (Nat, Nat) : Bool **reqs equality** (==)
function _!=_ (Nat, Nat) : Bool
endsig NAT

F.3 INT

The `INT` type represented signed integers of arbitrary precision. As the `NAT` type, it is a built-in type, not defined by constructors. Values are denoted by signed decimal numbers.

signature INT **is**
type Int **is**
<decimal digit sequence>
| -<decimal digit sequence>
endtype
function bin (<binary digit sequence>) : Int
function oct (<octal digit sequence>) : Int
function hex (<hexadecimal digit sequence>) : Int
function succ (Int) : Int
function pred (Int) : Int
function minus (Int) : Int
function _+_ (Int, Int) : Int
function _-_ (Int, Int) : Int
function _*_ (Int, Int) : Int
function _/_ [ZERO_DIVISION] (Int, Int) : Int
function rem [ZERO_DVISION] (Int, Int) : Int
function pow (Int, Int) : Int
function max (Int, Int) : Int
function min (Int, Int) : Int
function abs (Int) : Int
function _>_ (Int, Int) : Bool
function _>=_ (Int, Int) : Bool
function _<_ (Int, Int) : Bool
function _<=_ (Int, Int) : Bool

function _==_ (Int, Int) : Bool **reqs equality** (==)

function _!=_ (Int, Int) : Bool

endsig INT

F.4 FRAC

The "`FRAC`" type represents rational numbers. We present its interface and provide an implementation using the User Language for E-LOTOS:

signature FRAC **is**

import INT

type Frac **is**

frac (num: Int, den: Int) **where** den > 0

endtype

function _+_ (Frac, Frac) : Frac

function _-_ (Frac, Frac) : Frac

function _*_ (Frac, Frac) : Frac

function _/_ [ZERO_DVISION] (Frac, Frac) : Frac

function pow (Frac, Int) : Frac

function max (Frac, Frac) : Frac

function min (Frac, Frac) : Frac

function abs (Frac) : Frac

function round (x: Frac) : Frac

```
(* this function returns the nearest integral value to x, except for
   halfway cases, which are rounded to the integral value larger in
   magnitude *)
```

function ceil (x: Frac) : Frac

```
(* this function returns the least integral value greater than or equal to x *)
```

function floor (x: Frac) : Frac

```
(* this function returns the greatest value less than or equal to x *)
```

function _>_ (Frac, Frac) : Bool

function _>=_ (Frac, Frac) : Bool

function _<_ (Frac, Frac) : Bool

function _<=_ (Frac, Frac) : Bool

function _==_ (Frac, Frac) : Bool **reqs equality** (==)

function _!=_ (Frac, Frac) : Bool

endsig FRAC

A tentative implementation is given below:

module FRAC **is**

```
import INT
function gd (m: Int, n: Int) is
   case
   n :: 0 -> m
   |  otherwise -> gd (n, m rem n)
   endcase
endfunc
function reduce (f: Frac) is
   let gd: Nat = gd (abs (f.num), abs (f.den)) in frac (f.num / gd, f.den / gd)
endfunc
 function minus (f: Frac) : Frac is
   frac (minus (f.num), f.den)
 endfunc
function _+_ (f1: Frac, f2: Frac) : Frac is
   reduce (frac (f1.num * f2.den + f2.num * f1.den, f1.den * f2.den))
endfunc
function _-_ (f1: Frac, f2: Frac) : Frac is
   reduce (frac (f1.num * f2.den - f2.num * f1.den, f1.den * f2.den))
endfunc
function _*_ (f1: Frac, f2: Frac) : Frac is
   reduce (frac (f1.num * f2.num, f1.den * f2.den))
endfunc
function _/_ [ZERO_DVISION] (f1: Frac, f2: Frac) : Frac is
   case f2 in
      frac (num:=0, ...) -> raise ZERO_DVISION
   |  otherwise -> reduce (frac (f1.num * f2.den, f2.num * f1.den))
   endcase
endfunc
function pow (f: Frac, p: Int) : Frac is
   case p in
      0 -> frac (1, 1)
   |  when p lt 0 -> pow (f, p + 1) / f
   |  when p gt 0 -> pow (f, p - 1) * f
   endcase
endfunc
function max (f1: Frac, f2: Frac) : Frac is
   if f1 >= f2 then f1 else f2
```

```
    endfunc
    function min (f1: Frac, f2: Frac) : Frac is
        if f1 >= f2 then f2 else f1
    endfunc
    function abs (f: Frac) : Frac is frac (abs (f.num), f.den) endfunc
    function round (x: Frac) : Frac is
        frac ((x.num / x.den) +
                (if (x.num rem x.den) ge (x.den / 2) then 1 else 0),
                1)
    endfunc
    function ceil  (x: Frac) : Frac is
        frac (  (x.num / x.den) +
                (if (x.num rem x.den) ge 0 then 1 else 0),
                1)
    endfunc
    function floor (x: Frac) : Frac is
        frac ((x.num / x.den), 1)
    endfunc
    function _>_  (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) > 0
    endfunc
    function _>=_ (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) >= 0
    endfunc
    function _<_  (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) < 0
    endfunc
    function _<=_ (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) <= 0
    endfunc
    function _==_ (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) == 0
    endfunc
    function _!=_ (f1: Frac, f2: Frac) : Bool is not (f1 == f2) endfunc
endmod FRAC
```

F.5 FLOAT

The "`Float`" type is a built-in type for floating-point numbers, together with the associated functions.

signature FLOAT **is**

type Float **is**

[+|-]<dds>.[<dds>] [E[+|-]<dds>]

endtype

function _+_ (Float, Float) : Float

function _-_ (Float, Float) : Float

function _*_ (Float, Float) : Float

function _/_ [ZERO_DIVISION] (Float, Float) : Float

function max (Float, Float) : Float

function min (Float, Float) : Float

function pow (Float, Float) : Float

function abs (Float) : Float

function sqrt (Float) : Float

function exp (Float) : Float `(* e^x *)`

function log (Float) : Float `(* log_10 x *)`

function sin (Float) : Float

function cos (Float) : Float

function tan (Float) : Float

function asin (Float) : Float

function acos (Float) : Float

function atan (Float) : Float

function sinh (Float) : Float

function cosh (Float) : Float

function tanh (Float) : Float

function asinh (Float) : Float

function acosh (Float) : Float

function atanh (Float) : Float

function pi : Float

function e : Float

function round (x: Float) : Float

```
(* this function returns the nearest integral value to x, except for
   halfway cases, which are rounded to the integral value larger in
   magnitude *)
(* returns the greatest value less than or equal to x *)
(* returns the nearest integral value to x, except halfway cases
```

```
  are rounded to the integral value larger in magnitude *)
 function ceil (x: Float) : Float
  (* this function returns the least integral value greater than or equal to x *)
 function floor (x: Float) : Float
  (* this function returns the greatest value less than or equal to x *)
 function _>_  (Float, Float) : Bool
 function _>=_ (Float, Float) : Bool
 function _<_  (Float, Float) : Bool
 function _<=_ (Float, Float) : Bool
 function _==_ (Float, Float) : Bool reqs equality (==)
 function _!=_ (Float, Float) : Bool
endsig FLOAT
```

F.6 CHAR

```
signature CHAR is
 import NAT + BOOLEAN
 type Char is ...
 endtype
 function toNat   (Char)   : Nat
 function toChar  (Nat)    : Char
 function tolower (Char)   : Char
 function toupper (Char)   : Char
 function isalpha  (Char)  : Bool
 function isdigit  (Char)  : Bool
 function isxdigit (Char)  : Bool
 function islower  (Char)  : Bool
 function isupper  (Char)  : Bool
 function isalnum  (Char)  : Bool
 function _>_  (Char, Char)      : Bool
 function _>=_ (Char, Char)      : Bool
 function _<_  (Char, Char)      : Bool
 function _<=_ (Char, Char)      : Bool
 function _==_ (Char, Char)      : Bool reqs equality (==)
 function _!=_ (Char, Char)      : Bool
endsig CHAR
```

F.7 STRING

```
signature STRING is
   import NAT + CHAR + BOOLEAN
   type string is ...
   endtype
   function length (string)  : Nat
   function concat (string, string)  : string
   function prefix (string, Nat)      : string
   function suffix (string, Nat)      : string
   function substr (string, Nat, Nat)         : string
   function index  (string, string)  : Nat   (* search from left *)
   function rindex (string, string)  : Nat   (* search from right *)
   function nth (string, Nat)         : Char
   function _>_  (string, string)     : Bool
   function _>=_ (string, string)     : Bool
   function _<_  (string, string)     : Bool
   function _<=_ (string, string)     : Bool
   function _==_ (string, string)     : Bool reqs equality (==)
   function _!=_ (string, string)     : Bool
   function toString (Char)  : String
endsig STRING
```

G History of this document

G.1 Previous proposals regarding the data part

Version 1.0 (December 1995): Document entitled "A Proposal for the Data Type Part of E-LOTOS Applicable to the Formal Description of OSI and ODP Standards". Input document [LG4] of the E-LOTOS interim meeting in Liège. Also available as technical report [SG95].

Version 1.1 (March 1996): "A Proposal for the Data Type Part of E-LOTOS Applicable to the Formal Description of OSI and ODP Standards". Annex A part 2 of the output document of the E-LOTOS interim meeting in Liège [Que96b]. This version differs from version 1.0 only by the correction of typos and English mistakes.

G.2 Previous proposals regarding the behaviour part

Version 1.0 (June 1994): Document entitled "Six improvements to the process part of LOTOS". Annex K of the Working Draft on Enhancements to LOTOS (ISO/IEC JTC1/SC21/WG1/N1349). Also Annex F of the Revised Working Draft (V2) on Enhancements to LOTOS (ISO/IEC JTC1/SC21 N10108). Also available as a technical report [Gar94b].

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.

Version 2.0 (December 1995): Document entitled "A wish list for the behaviour part of LOTOS". Input document [LG5] of the E-LOTOS interim meeting in Liège. Also available as a technical report [Gar95b].

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

Version 2.1 (March 1996): Document entitled "A wish list for the behaviour part of LOTOS". Annex F Part 2 of the output document of the E-LOTOS interim meeting in Liège [Que96b]. Both the Committee's decisions and the comments resulting from discussions in Liège have been integrated in the new version 2.1. The following changes have been brought:

- The present Annex was added, to keep track of changes to this document
- All the sections of the previous version 2.0 has been changed into sub-sections of Section 2 of this document.
- In Section 2.2: the wording of the proposal has been revised.
- In Section 2.4: regarding the translation of the usual "**case**" into the general "**case**", a mistake signaled by Alan Jeffrey has been corrected.
- In Section 2.7: the ring example was wrong and has been corrected. A missing static semantics constraint has been added for renamings. A few wrong indexes have been fixed
- In Section 2.9: the parallel composition operators have been simplified.
- In Section 2.11: this section has been added. A unique parallel operator was defined as the main paradigm (this operator was previously described in a remark at the end of Section 10).
- In Section 2.12: the "n among m" synchronization is now introduced as a shorthand for the general parallel operator presented in Section 2.11

- In Section 2.13: this section has been fully rewritten. Exceptions are now presented before sequential composition since it is easier to present exceptions first and sequential composition as a shorthand for exceptions. Exception identifiers are now distinct from gate identifiers. The syntax of the "**trap**" operator was changed (the reasons are explained in a remark). The issue of exception declaration and scope rules have been solved. Exceptions are added to process definitions and instantiations. The parallel composition operator was modified to integrate exceptions (following a comment by Alan Jeffrey that exceptions should not be synchronized by default). Most of the remarks "for further study" have been processed. Following a discussion with Guy Leduc, the translation rule from "`[>`" to "**trap**" was corrected.
- In Section 2.13: the two previous Sections 12 and 13 (syntactic and semantic unification of the "`;`" and "`>>`" operators) have been merged into a single one, since this separation generated much confusion during the Liège meeting. In the new Section 2.13, the key ideas are (hopefully) better explained.
- In Section 2.18: The proposal for abbreviating formal gate parameter lists (former Section 18.1) has been removed, as decided by the Committee.
- In Section 2.19: The proposal for abbreviating formal value parameter lists (former Section 19.1) has been removed, as decided by the Committee.
- In Section 3 and 4: These Sections have been added. They give a formal treatment of the enhancements described in Section 2.
- In Annex B: This Annex was added to keep track of Brinksma's proposal for unifying the "`;`" and "`>>`" operators (previously in Section 12).
- In Annexes C and D: Due to the Committee's decision to remove "**where**" clauses in process definitions (Section 2.16) and due to the changes in Sections 2.18 and 2.19, the fragments of E-LOTOS programs given in these Annexes have been "flattened" to remove nested processes.

G.3 Previous proposals regarding the gate typing

To be completed.

G.4 History of this document

Version 1.0 (May 1996): Document entitled "French-Romanian Integrated Proposal for the User Language of E-LOTOS". Input document of the E-LOTOS interim meeting in Kansas City. This document attempts to merge the above proposals for the data part, the behaviour part and gate typing into a single, coherent proposal, which covers abstract syntax and static semantics, and provides one possible dynamic semantics.

It benefits from the discussions with Alan Jeffrey, Guy Leduc, Charles Pecheur, and Ricardo Pena during a short-term scientific mission (April, 22–26, 1996) supported by the European Commission under Cost 247 Action and European-Canadian project Eucalyptus-2.

As regards the data part, the following changes have been brought with respect to [Que96b, Annex A, Part 1] ("A Proposal for the Data Type Part of E-LOTOS Applicable to the Formal Description of OSI and ODP Standards", version 1.1):

- The Full User Language (Section 4) supersedes the data user language.

- Exceptions in the data part have been enhanced to support value parameters. The notion of exception type was added, and the declaration of exceptions was modified to take exception types into account. The "**raise**" operator was extended to support actual parameters for exceptions.
- The scoping rules for exception identifiers (previously the same as in SML) have been aligned with the corresponding rules in the behaviour part. The "**trap**" operator was extended to support formal parameters for exceptions. Function instantiations have been extended with exception identifiers.
- The distinction previously made between "expressions" and "instructions" (which was criticized during the Liège meeting) was lifted. Exceptions and instructions are now unified due to the use of "imperative-like" language features (operators "`:=`", "`;`", and "**local**"). The class *Instr* has been removed.

  As a consequence, the "**return**" operator previously used for assigning "**out**" parameters was replaced with the "`:=`" operator. The "`eval`" operator has been removed, as it is now obtained by combining function instantiation and the "`;`" operator. Also, the syntax of function instantiation for "out" parameters has been modified with the "`=: &`" keyword. The two different syntaxes for function declarations have been unified.
- The "E `match` P" operator was introduced.
- For symmetry with the behaviour part, the "**loop**" and "**continue**" operators have been introduced in the data part.
- Some syntactical restrictions of data language have been lifted and integrated in the static semantics.
- Pragmatic restrictions have been introduced to ensure that all overloaded functions with the same number of formal arguments have the same names for their formal arguments. Therefore, neither formal argument names nor exceptions cannot be used to solve overloading.
- The static semantics of the data part, previously defined using attribute grammars, is now expressed as a set of Plotkin rules.
- The dynamic semantics of the data part, previously defined in a denotational way, is now an operational one.

As regards the behaviour part, the following changes have been brought with respect to [Que96b, Annex F, Part 2] ("A wish list for the behaviour part of LOTOS", version 2.1):

- The Full User Language (Section 4) supersedes the behaviour language of [Que96b, Annex F, Part 2].
- The concept of gate typing has been introduced in the behaviour part with respect of the proposal made in [Gar95a]. In particular, the declarations now include "**channel**" types. All gate declarations are typed with a channel.
- Gates and exceptions (which formerly were identical objects and later distinct objects) are unified again. Process declarations, process instantiations, parallel composition operators, "**raise**", "**trap**", and "**renaming**" operators have been modified accordingly. Exception types are now syntactic sugar for non-overloaded channels.
- To ensure symmetry with the data part, and to solve the many drawbacks of LOTOS sequential composition, the "**local**" and "`:=`" operators have been added to the behaviour part. Similarly , an "`:=` **any**" operator was added to express non-deterministic choice.

- Former LOTOS operators "**let**", "**choice**", "`[]`", and action-prefix are now obtained as derived cases of the three aforementioned operators.
- The keyword "**sel**...**endsel**" have been replaced by "**alt**...**endalt**" to avoid confusion with "**seq**...**endseq**" and to remind about OCCAM.
- Variable assignments in "**exit**" operands, i.e., "**exit** (V := E)", are no longer permitted, since the same effect can be obtained by writing "V := E ; **exit** (V)".
- According to the"uniform reference" principle, we propose a similar syntax for LOTOS gates and process instantiations. This is similar to what exists in other FDTs, namely ESTELLE and SDL. This proposal is more powerful than in LOTOS, since patterns are allowed in offers. This will allow an easier mapping of IDL interfaces to LOTOS gates. It will also allow action refinement (i.e., replacement a LOTOS gate with a process instantiation). Additionally, it solves a syntactic ambiguity problem between the new sequential composition operator and the former action prefix (a problem pointed out by Alan Jeffrey).
- The "**loop**" and "**continue**" operators, which were already proposed in the rationale part, have been explicited in the abstract syntax.
- The syntax of the "**rename**" operator has been simplified.
- The former transition function from "user language" to "core language" was divided into two successive translation functions, using an intermediate language, the "reduced user language". The first one performs transformations based only upon syntax and declaration information; the second one performs type-checking (including overloading solving) and functionality computation. This organization leads to a simplified presentation.
- The rules for functionality computation have been adapted to the reduced user language. They are now included in the static semantics.
- The dynamic semantics of the parallel and renaming operators was corrected.

References

[Arn92] André Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Masson, 1992.

[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.

[BL95] E. Brinksma and G. Leih. *Enhancements of LOTOS.* In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 453–466. Kluwer Academic Publishers, 1995.

[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.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[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.

[Gar95a] 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.

[Gar95b] Hubert Garavel. A Wish List for the Behaviour Part of E-LOTOS. Rapport SPECTRE 95-21, VERIMAG, Grenoble, December 1995. Input document [LG5] of the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

[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 [LG3] 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 N10108 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[NS90] Xavier Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP: Theory and Application. Rapport SPECTRE C26, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, December 1990.

[NS94] Xavier Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994 1994.

[QA92] J. Quemada and A. Azcorra. Structuring Protocols with Exception in a LOTOS Extension. In *Proceedings of the 12th IFIP International Workshop on Protocol Specification, Testing and Verification (Orlando, Florida, USA)*, Amsterdam, June 1992. IFIP, North-Holland.

[Que96a] Juan Quemada, editor. Revised Working Draft on Enhancements to LOTOS (V2). ISO/IEC JTC1/SC21/WG7 N10108 Project 1.21.20.2.3. Output document of the Ottawa meeting, January 1996.

[Que96b] Juan Quemada, editor. Revised Working Draft on Enhancements to LOTOS (V3). ISO/IEC JTC1/SC21/WG7 N1053 Project 1.21.20.2.3. Output document of the Liège meeting, March 1996.

[RS90] Valérie Roy and Robert de Simone. Auto/Autograph. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 477–491. AMS-ACM, June 1990.

[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. AFNOR CGTI/CN 21 F 2503.