LOTOS NT User Manual

Mihaela Sighireanu

with updates by Alban Catry, David Champelovier, Hubert Garavel, Frédéric Lang, Guillaume Schaeffer, Wendelin Serwe, and Jan Stöcker

Release 2.6 — February 21, 2008

Foreword

The present User Manual of Lotos NT comes with the release 2.6 of the compiler Traian. It completely describes the syntax and the informal semantics of the sub-set of Lotos NT currently supported by Traian.

Status This is a partial implementation; it does not fully implement the behaviour and module parts of Lotos NT. The "†" symbol is used to mark the features of Lotos NT which are not yet supported by Traian.

Availability The complete distribution for Traian is available on the web at the address `http://www.inrialpes.fr/vasy/traian`. Please report feedback and bugs to `traian@inrialpes.fr`.

Acknowledgments The author thanks people who helped in writing this manual, either through their ideas or their comments about the different versions of this manual. I owe thanks to Frédéric Lang, and also to Fabrice Baray, Claude Chaudet, Hubert Garavel, Marc Herbert, Radu Mateescu, and Bruno Vivien.

Contents

1 Introduction **9**
- 1.1 Background . . . 9
- 1.2 Goals . . . 11
- 1.3 Main Concepts . . . 11
- 1.4 LOTOS NT versus E-LOTOS . . . 12
- 1.5 Manual Structure . . . 13

2 Basic mathematical concepts and notation **15**
- 2.1 General . . . 15
- 2.2 Backus-Naur Form . . . 16
- 2.3 Description of the Syntax . . . 17
- 2.4 Data values . . . 17
- 2.5 Structured Labelled Transition System . . . 17

3 Lexical Structure **19**
- 3.1 ISO Latin-1 Character Set . . . 19
- 3.2 Input Elements and Tokens . . . 20
- 3.3 Comments . . . 20
- 3.4 Includes . . . 20
- 3.5 Identifiers . . . 21
- 3.6 Special Identifiers . . . 21
- 3.7 Keywords . . . 22
- 3.8 Literals . . . 22
  - 3.8.1 Integer Literals . . . 22
  - 3.8.2 Floating-Point Literals . . . 23
  - 3.8.3 Characters . . . 23
  - 3.8.4 String Literals . . . 24
- 3.9 Operators . . . 25

4 Types **27**
- 4.1 Type Definition . . . 27
- 4.2 Predefined Operations . . . 28
- 4.3 Predefined Types . . . 29
  - 4.3.1 The boolean type . . . 30
  - 4.3.2 The natural type . . . 30
  - 4.3.3 The integral type . . . 30
  - 4.3.4 The floating point type . . . 31
  - 4.3.5 The character type . . . 31

4.3.6 The string type 32
4.3.7 The “none” type 32
4.3.8 The “time” type 32
4.4 Derived Types 33
4.4.1 Enumerated types 33
4.4.2 Scalar and simple types 34
4.4.3 Record types 34
4.4.4 Sets 35
4.4.5 Lists 35
4.5 Subtypes† 37
4.6 Arrays† 37
4.7 External Types and Pragmas 37

5 Expressions, Statements, and Functions 39
5.1 Constants 39
5.2 Value expressions 40
5.2.1 Variables 41
5.2.2 Constructor application 41
5.2.3 Function application 42
5.2.4 Field selection 42
5.2.5 Field update 43
5.2.6 Unambiguous Expressions 43
5.3 Patterns 44
5.4 Statements 45
5.4.1 Value return 47
5.4.2 Null Statement 47
5.4.3 Assignment 47
5.4.4 Sequential Composition 47
5.4.5 Variable declaration 47
5.4.6 Case statement 48
5.4.7 If statement 49
5.4.8 Iteration Statements 50
5.4.9 Exceptions and their handling 52
5.5 Functions 54
5.5.1 Function definition 54
5.5.2 Function call 56

6 Behaviour Expressions and Processes 59
6.1 Basic Behaviours 59
6.2 Regular Behaviours 60
6.2.1 Variable declaration 60
6.2.2 Conditional behaviour 60
6.2.3 Iteration behaviour 61
6.2.4 Exception raising and handling 62
6.3 Assignment and Procedure Call 62
6.4 Communication and Actions 63
6.5 Sequential Behaviours 65
6.6 Signaling 65
6.7 Processes Definition and Instantiation 66
6.7.1 Process definition 66

6.7.2 Process instantiation . . . 67
6.8 Non-deterministic Behaviours . . . 68
6.8.1 Operator “[]” . . . 68
6.8.2 Operator choice over values . . . 69
6.9 Concurrency . . . 70
6.9.1 Synchronization operator . . . 70
6.9.2 Full synchronization operator . . . 71
6.9.3 Interleaving operator . . . 71
6.9.4 Generalized parallel operator . . . 72
6.9.5 Parallel over values operator . . . 74
6.10 Hiding . . . 74
6.11 Renaming . . . 76
6.12 Unambiguous Behaviours . . . 77

7 Modules 79
7.1 Generalities . . . 80
7.2 Interfaces . . . 81
7.2.1 Interface declaration . . . 81
7.2.2 Interface expression . . . 81
7.2.3 Equation declaration . . . 83
7.3 Modules . . . 84
7.3.1 Module declaration . . . 84
7.3.2 Module expression . . . 85
7.3.3 External modules . . . 86
7.4 Generic Modules . . . 86
7.4.1 Generic module declaration . . . 86
7.4.2 Generic module instantiation . . . 87
7.5 Specification . . . 89
7.6 Importation Rules . . . 89

A Syntax Summary 91
A.1 Syntax of the module part . . . 91
A.2 Syntax of the data part . . . 93
A.3 Syntax of the behaviour part . . . 97

Bibliography 101

Index 105

Chapter 1

Introduction

"*Formal description techniques (FDT) are methods of defining the behaviour of an (information processing) system in a language with formal syntax and semantics, instead of a natural language as English.*"
ISO-8807

In the following section the origin and the evolution of FDTs is discussed, especially Lotos. The objectives that the new generation of FDTs must satisfy are considered. The main concepts of E-Lotos are presented. Finally the structure of the document is explained.

1.1 Background

The origins of FDTs are the activity of international organizations as Iso and Ccitt around the open architecture Osi. Due to their complexity, the description of services and protocols of the communication systems composing the Osi architecture needed best methods than natural language and state machines. For this reason Iso and Ccitt developed in the 80's three formal description techniques: Estelle, Lotos, and Sdl.

Lotos was developed by FDT experts from Iso during the years 1981-1988. The objectives of its design follow strictly the main general objectives defined for FDTs:

- *expressive*: Lotos is able to define both protocol and service descriptions of the seven layers of Osi.
- *well-defined*: Lotos has a formal mathematical model suitable for the analysis of descriptions supported by the testing of an implementation for conformance.
- *well-structured*: Lotos offers many means for structuring of specification.
- *abstract*: Lotos is independent from the methods of implementations and offers means for abstraction of irrelevant details.

As a design choice, Lotos consists of two "orthogonal" sub-languages:

The data part of Lotos is dedicated to the description of data structures. It is based on the well-known theory of algebraic abstract data types [Gut77], more specifically on the ActOne specification language [dMRV92].

The control part of Lotos is based on the process algebra approach for concurrency, and appears to combine the best features of Ccs [Mil89] and Csp [Hoa85].

Lotos has been applied to describe complex systems formally, for example: the service and protocols for the Osi transport and session layers [ISO89b, ISO89a, ISO92b, ISO92c], the Ccr[1] service and protocol [ISO95b, ISO95a], Osi Tp[2] [ISO92a, Annex H], Maa[3] [Mun91], Ftam[4] basic file protocol [LL95], etc. It has been mostly used to describe software systems, although there are recent attempts to use it for asynchronous hardware descriptions [CGM$^+$96].

A number of tools have been developed for Lotos, covering user needs in the areas of simulation, compilation, test generation, and formal verification.

Nevertheless, the three FDTs, including Lotos, actually show their limitation for several reasons:

- Some design choices must be revised in order to respond to criticism of users. For example, the abstract data types used in Lotos and Sdl do not satisfy users.
- The new communication protocols like those of high flow network (e.g., Atm) or multimedia protocols need the specification of real-time constraints. None of the three FDTs allows to express all needed quantitative temporal constraints.
- The development of new architectures like Odp[5] or Corba[6] call into question the Osi model and its static architecture. The model chosen is more dynamic and the mobility is important.

For these reasons Iso/Iec begun a revision of the Lotos standard in 1993 in the frame of Odp activity group. The revised language is called E-Lotos (for Extended-Lotos). The enhancements of Lotos should remove known limitations of the language concerning expressiveness, abstraction and structuring capabilities, user friendliness. A non-exhaustive list of such undesirable characteristics is given below:

- Although having a strong mathematical basis, the abstract data types need a good background from the part of users. This prevents the use of the language by a large public, restricting it to an "expert" public.
- Lotos is able to describe only temporal ordering, for example "the sending of a message is followed by its reception". However, one needs to express quantitative time requirements like "the sending of a message is followed after 5 seconds by the message reception".
- In the control part, the value passing is done in a pure functional style. Despite its proper semantics, this feature adds cumbersome constraints for structuring the specification. For this reason, a lot of case studies are done using "Basic Lotos", *i.e.*, Lotos without values.

[1]Commitment, Concurrency, and Recovery
[2]Distributed Transaction Processing
[3]Message Authentication Algorithm
[4]File Transfer, Access, and Management
[5]Open Distributed Processing
[6]Common Object Representation Brooker Architecture

1.2 Goals

This section lists a number of qualities which, in our opinion, the new generation of FDT languages should have.

The first of them is that E-Lotos must be a useful and pleasant *tool* for behaviour description and analysis, and not a set of more or less awkward constraints.

This language must be *easy to learn*, which implies that its constructions have a *well defined, non-ambiguous semantics*, and that it respects (in the limits of the semantics) most rules and habits of the users. Because programming languages are intuitive and their concepts largely used, providing the language with algorithmic features is a mean to accomplish this goal.

The language should also provide as much as possible *description safety and reliability*, while remaining very *versatile*. As many errors as possible must be detected at compile time.

The language should provide maximal *expressiveness*. For example, the use of Lotos in several case studies showed that the operators and the concepts of the language are not able to express self interruption of behaviour [QA92], deterministic control passing between processes [GH93], or nets of processes [Bol90]. E-Lotos should fill these gaps. Expressiveness also concerns the means for describing real-time aspects. Actually, there exists several extensions of Lotos with real-time operators: ET-Lotos [LL97], RT-Lotos [Cd95]. These extensions form a strong basis for the definition of real-time aspects of E-Lotos. As an extension of Lotos, the language should provide mean for upward compatibility: a translation of Lotos constructs in E-Lotos should be provided.

The language should allow the *modular* description of systems and description *re-usability*.

In the context of the Odp group at Iso/Iec, the language should provide means for an *easy interface* with external description or programming languages developed by this group, e.g., Idl. Also, it should remain independent from, but *easily translatable* into most implementation languages (first targets being C, Ada, Java). The accomplishment of this goal will provide a good platform for tools developers, and so a possible large distribution of the language.

The language should provide constructs offering opportunities for an *optimal analysis* by tools.

Last but not least, the language must be *simple*.

1.3 Main Concepts

This section presents the main concepts of E-Lotos, together with a short justification of their introduction in the language. Those justifications are related to the goal listed in the previous section.

First of all, the E-Lotos main feature is *concurrency*. It is a mean for description of concurrent (parallel) evolution of systems and their communication. The systems are composed in parallel using a Csp-like [Hoa78] operator. The base mechanism for communication is the *rendezvous* on communication points called *gates*. The communication allows the exchange of values. This is the only mean for interaction between concurrent systems because their memory spaces must be disjoint. The language provides several mechanisms for concurrency: interleaving, binary and n-ary synchronization, network synchronization, coroutine mechanism. This is a first step towards language expressiveness.

E-Lotos is a description language supporting non-determinism. Both internal and external [Hoa78] non-determinism is provided. By difference with Lotos, E-Lotos provides also deterministic choice constructs by means of “if-then-else” and “case” statements. The introduction of these constructs touches both easy to use and optimal analysis requirement.

The language provides means for *real-time* descriptions. All operators of the language have an intuitive time semantics. The time domain may be defined by the user with respect to a proper semantics. So the time domain may be dense or discrete.

In the same frame of expressiveness, E-Lotos supports *exception handling* in order to deal with abnormal conditions. The exceptions are modeled by signals.

The language is *strongly typed*, a necessary condition for description safety. All objects in a description must be typed. *Type checking* is performed on any E-Lotos description in order to detect, at compile time, most inconsistencies and errors. Basic types include integers, reals, booleans, strings, etc. User defined types may be defined by using type constructors. This provides means for defining most usual types: enumeration types, records, unions, sets, lists. Types may be recursive. Also, types and functions may be specified into an external language.

E-Lotos remains a *functional* language in its semantics, Although it supports assignment of variables. This is a step toward user friendliness on the one hand, and interfacing with external languages, on the other hand.

Modularity is a basic feature of E-Lotos. Constants, types, functions, and processes may be defined in separate *modules*. The modules support the definition of local objects (constants, types, functions, and processes). The visibility of local objects is specified by means of module *interfaces*. Modules may be combined by *importation*. Another important feature for re-usability purposes is *genericity*. Generic modules provide means for parameterizing modules with constants, types, functions, and processes. As in Lotos, the dynamic semantics of behaviours and expressions are given only for fully instantiated modules.

1.4 LOTOS NT versus E-LOTOS

Lotos NT is the language supported by the Traian compiler. It follows the main concepts of E-Lotos and offers other features, in order to provide versatility, compilation and verification efficiency.

[Sig99] exposes the main differences between Lotos NT and E-Lotos. We cite only two examples:

- In Lotos NT, functions names may be *overloaded* as in Ada [WWF87], *i.e.*, two or more functions may have the same name provided they have different *profiles* (list of parameters types and result type). This is a useful feature because it improves the semantic consistency of a Lotos NT specification—two similar operations on different types need not have different names—and the semantic consistency of Lotos NT predefined functions themselves. Also, the compatibility with ActOne is ensured.

- In Lotos NT, functions and processes may have input, output, and input/output parameters as in Ada. This provides means for returning several results and for easy interfacing with languages as Idl.

- The style of the Lotos NT language is fully imperative in syntax and semantics, unlike E-Lotos which has functional semantics.

1.5 Manual Structure

This manual gives an informal definition of the LOTOS NT language. A formal definition may be found in [Sig99].

Chapter 2 presents the mathematical notations and concepts used. Chapter 3 presents the lexical structure of the language. The language constructs are presented bottom-up, in order to make the language easier to learn. We begin by presenting types and type declarations in chapter 4. The language of data is presented in chapter 5. It contains data expressions, statements, and function declarations. Chapter 6 presents the core of the language: behaviour expressions and process declarations. Modules are presented in chapter 7.

Each section of the chapters defining the language presents language constructs in the following order:

- the goals and the rationales of the construct;
- its abstract syntax;
- its intuitive and its formal, static, and dynamic semantics;
- some examples of its use.

In this presentation, we use the "†" symbol to mark the features of the language which are not yet supported by the TRAIAN compiler.

Annex A presents the full syntax of the language.

We tried to present the information in a strictly linear order. However, where it is not possible to do so, we signal forward references.

Chapter 2

Basic mathematical concepts and notation

2.1 General

This section contains a list of basic mathematical concepts and related notations used in the remainder of the document.

$\overset{def}{=}$	is defined as.
iff	if and only if, *i.e.*, double implication.
$\{a, b, c, ...\}$	the *set* made up of elements $a, b, c, ...$. The order in which the elements are listed is immaterial.
$\emptyset$	the *empty set*.
$x \in A$	x is an *element* of the set A.
$x \notin A$	x is *not* an *element* of the set A.
$A \subseteq B$	A is a *subset* of B.
$A \times B$	the *Cartesian product* of A with B, *i.e.*, the set of all ordered pairs $< a, b >$ such that $a \in A$ and $b \in B$.
$A_1 \times A_2 \times ... \times A_n$	the *generalized Cartesian product* of $A_1, A_2, ..., A_n$, *i.e.*, the set of ordered tuples $< a_1, a_2, ..., a_n >$, such that $(\forall i) a_i \in A_i$.
$\{x \in A \mid Q(x)\}$	the set which contains only all those elements of A which satisfy the property Q.
$a_1, ..., a_n$	the finite (or empty) list (or sequence, or n-tuple) made up of the *elements*, or *components* $a_1, ..., a_n$. Unlike sets, lists may contain more than one instance of the same element, since elements are distinguished by their position in the ordering of the list; the length of the list is n;
$<>$	the *empty* list has no elements, its length is 0;
$a_0, ..., a_n$	the non-empty finite list made up of the elements $a_0, ..., a_n$; the length of the list is $n + 1$;

$\overline{a}$	the non-empty finite list made up of the elements $a_0, ..., a_n$; the length of the list is $len(\overline{a})$; a *record* is a n-tuple of which each element is *labelled* with a unique label. If *lab* is the label of element x of record y, then $y.lab$ denotes x.
$R \subseteq A \times B$	R is a *binary relation* between A and B, *i.e.*, a set of elements of $A \times B$; the *domain* of R is defined as $\{a \in A \mid \exists b \in B. < a, b > \in R\}$; the *range* of R is defined as $\{b \in B \mid \exists a \in A. < a, b > \in R\}$;
$\{\}$	the empty relation;
$f : A \rightarrow B$	f is a (*partial*) *function* (*finite map*) from A to B, *i.e.*, f is a binary relation between A and B such that for each $a \in A$ there exists at most one $b \in B$ such that $< a, b > \in f$; the domain of f is denoted by $\mathrm{Dom}(f)$; the range of f is denoted by $\mathrm{Ran}(f)$; if $< a, b > \in f$ then f is *defined* for a, also written $f(a) = b$ or $a \mapsto b$; the function f is total iff it is defined for all $a \in A$; a function $f : A \rightarrow B$ is *injective* iff, for all a_1, a_2 in the domain of f, $f(a_1) = f(a_2)$ implies that $a_1 = a_2$;
$f : A_1 \times A_2 \times ... \times A_n \rightarrow B$	the function from the Cartesian product $A_1 \times A_2 \times ... \times A_n$ to B; the function *arity* maps f to the number n of terms of the Cartesian product.

2.2 Backus-Naur Form

The meta-language used in this manual to specify the syntax is based on Backus-Naum Form (BNF). A BNF description of a language L is given by a set of *productions*, or re-write rules. The meta-symbols used to compose rewrite rules are listed in Table 2.1.

Meta-symbol	Name	Pronunciation
`"xyz"`	terminal symbol	xyz
`abc`	nonterminal symbol `abc`	(nonterminal) abc
`::=`	rewrite symbol	is defined to be
`|`	alternation symbol	or, alternatively
`[...]`	option operator	0 or 1 instances of
`{...}`	repetition operator	0 or more instances of
`;`	semi-colon	end of BNF rule

Table 2.1: Meta-language symbols

A *terminal symbol* is a symbol that appears literally in L. A *nonterminal symbol* is a symbol that denotes a syntactic construct of L (which is ultimately represented by a string of terminal symbols).

A rewrite rule has the form:

```
<nonterminal-symbol> ::= meta-expression ;
```

where the meta-expression is an expression constructed using terminal and nonterminal symbols, and the operators listed in Table 2.1 except `::=` and `;`. Adjacent terminal or/and nonterminal symbols occurring in a meta-expression denote the lexical concatenation of the texts they ultimately represent. Concatenation respects the rules given in 3.

A rewrite rule is interpreted as follows: the nonterminal symbol of the left-hand side can be replaced by any one of the of the sequences separated by the alternation symbol.

All operators (including implicit concatenation) have precedence order over the alternative operator.

2.3 Description of the Syntax

Descriptions of concrete syntax give formal rules to be implemented by a parser for the language. Concrete syntax descriptions obey to constraints dictated by the implementation on a computer. For instance, in order to avoid ambiguities and to provide a simple parser, the grammar must be LALR(1).

However, the purpose of this document is to present the syntax to the user of the language. In order to be more easily readable, we can abstract out some implementation details, and provide a more informal presentation of the concrete syntax, using meta level syntactic facilities. A human will understand the description better and faster than if written in a language designed for a machine. It uses type-setting conventions which facilitate the user reading. The conventions used for the presentation of the syntax are the following:

- terminals are represented using bold face;
- the special symbols are represented using teletype font. Note the difference between the special symbols "`[`" and "`]`" and the (mathematical style) symbols "[" and "]" used to express optional syntactic clauses in BNF.
- a non-empty list is represented like "a_0`,`...`,`a_n", *i.e.*, with the indexes starting at 0. The possibly empty lists are indexed from 1, *i.e.*, "a_1`,`...`,`a_n".

  More precisely, the BNF equivqlent of "a_0`,`...`,`a_n" is "$a\{$`,`$a\}$", while the BNF equivalent of "a_1`,`...`,`a_n" is "$[a\{$`,`$a\}]$".

2.4 Data values

A *data domain* D is a set of sets; the elements of D are referred to as data carriers.

A special data carrier is the time data carrier. It is denoted by "**time**".

NOTE: The domain of a type is a data carrier.

2.5 Structured Labelled Transition System

A *structured labelled transition system* is a 5-tuple $\langle Q, L \cup \{\mathbf{i}\}, D, T, q_{init}\rangle$ where:

- L is a set of *labels*;
- $\mathbf{i} \notin L$ is an *internal event*;
- D is a data domain;
- $\langle Q, A, T, q_{init}\rangle$ is a labelled transition system, where

$$A \overset{def}{=} \{\mathbf{i}\} \cup \{G\ N \mid G \in L, N \in D\} \cup \{d \mid d \in \mathbf{time}\}$$

To emphasize the particular domain values D of a structured labelled transition system, this system is also referred to as a labelled transition system *over* D.

Chapter 3

Lexical Structure

This chapter presents[1] the lexical conventions of Lotos NT.

Lotos NT programs uses the ISO Latin-1 (8859.1) character set.

The characters resulting from the lexical translations are reduced to a sequence of input elements (§ 3.2, p. 20), which are spaces, comments (§ 3.3, p. 20), and tokens. The tokens are: identifiers (§ 3.5, p. 21), keywords (§ 3.7, p. 22), literals (§ 3.8, p. 22), and operators (§ 3.9, p. 25) of the Lotos NT syntactic grammar.

3.1 ISO Latin-1 Character Set

The ISO Latin-1 character set is divided into:

- alphabetic characters (letters), made of ASCII [2] characters (octal codes #101–#132) and other ISO Latin 1 characters (octal codes #300–#377). See Table 2 of ISO/IEC DIS 14750.

```
LETTER ::= #101..#132 ;

LETTER_WITH_ACCENT ::= #300..#377 ;

ALPHABETIC_CHARACTER ::= LETTER | LETTER_WITH_ACCENT ;
```

- digits, *i.e.*, characters from "0" to "9". See Table 3 of ISO/IEC DIS 14750.

```
DIGIT ::= "0".."9" ;
```

- spaces and formating characters, which include blanks, horizontal and vertical tabs, newlines, form feeds. See Table 5 of ISO/IEC DIS 14750.

```
SPACE  ::= HT | NL | FF | SP | LF | CR ;
```

NOTE: IDL considers also BEL, BS, but not SP.

[1] This section is an adaptation of *The ISO/IEC DIS 14750*, Section 4 (IDL Syntax and Semantics); it differs in the list of legal keywords and punctuation.

[2] ASCII (ANSI X3.4) is the American Standard Code for Information Interchange.

Except for comments, identifiers, and the contents of character and string literals, all input elements in a Lotos NT specification are formed only from ASCII characters (or escapes which result in ASCII characters).

3.2 Input Elements and Tokens

The input characters and line terminators are reduced to a sequence of input elements. Input elements which are not blank spaces or comments are tokens. Tokens are the terminal symbols of the Lotos NT syntactic grammar.

```
Input ::= [ InputElement { InputElement } ] ;

InputElement ::= SPACE | Comments | Token ;

Token ::= IDENTIFIER | Keyword | Literal | Operator | Separator ;
```

There are four classes of tokens: identifiers, keywords, literals, operators, and other separators. Blank spaces and comments are ignored except as they serve to separate tokens. Some blank space is required to separate otherwise adjacent identifiers, keywords, and literals.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

3.3 Comments

Lotos NT defines two kinds of comments:

- `(* text *)` A Lotos comment; all the text from the ASCII characters `(*` to the ASCII characters `*)` is ignored.
- `-- text` A single line comment: all the text from the ASCII characters `--` to the end of the line is ignored.

Comments do not nest. The comment characters `--`, `(*`, and `*)` have no special meaning within a `--` comment or within a `(*` comment. Comments may contain alphabetic, digit, graphic, and space (but not newline) characters.

Comments are not part of the Lotos NT description. They may be inserted anywhere between two other lexical units or left out, except when they play the role of separators.

Zero or more separators may occur between any two consecutive tokens, before the first token, or after the last token of the Lotos NT text.

There shall be at least one token separator between any pair of consecutive tokens if the concatenation of their texts change their meaning.

3.4 Includes

The `library ... end library` sequence allows to include files in the source code. This feature enables to write Lotos NT descriptions in separate files, so as to improve modularity.

The include mechanism works like the `#include` in C language: a file can be included anywhere in the source code, and the lexical analyser is in charge of replacing the sequence by the content of the included file.

```
Include ::= Library { SPACE | Comments }
                """ Filename """ { SPACE | Comments }
                { "," { SPACE | Comments } Filename { SPACE | Comments } }
            End { SPACE | Comments } Library ;

Library ::= ("l" | "L") ("i" | "I") ("b" | "B") ("r" | "R")
            ("a" | "A") ("r" | "R") ("y" | "Y") ;

End ::= ("e" | "E") ("n" | "N") ("d" | "D") ;
```

`Filename` is the path to the included file. It can be either absolute or relative to the current working directory.

Several files can be included in the same `library ... end library` sequence. In this case, the files will be included in the same order as they appear in the sequence.

3.5 Identifiers

An identifier is an unlimited-length sequence of alphabetic, digit, and underscore ("_") characters. The first character must be an alphabetic character.

```
IDENTIFIER ::=  ALPHABETIC_CHARACTER { NORMAL_CHARACTER | "_" } ;

NORMAL_CHARACTER  ::= DIGIT | ALPHABETIC_CHARACTER ;
```

In LOTOS NT, identifiers are not case-sensitive. In a given declaration scope, two identifiers that differ only in the case of their characters are considered redefinitions of one another: they will collide and yield a compilation error. When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. TBL 2 in (ISO/IEC DIS 14750) defined the equivalence mapping of upper- and lower-case letters.
- The comparison does *not* take into account equivalences between diagraphs and pairs of letters (e.g., "æ" and "ae" are not considered equivalent) or equivalences between accented or not accented letters (e.g., "à" and "a" are not considered equivalent).
- All characters are significant.

3.6 Special Identifiers

In order to allow a more intuitive notation for the different mathematical operators, a special class of identifiers is introduced. The special identifiers are built as follows:

```
SPECIAL_CHARACTER ::= "%" | "&" | "*" | "+" | "-" | "." | "/" | ">" | "="
                    | "<" | "@" | "\" | "^" | "~" | "{" | "}" ;

SPECIAL_IDENTIFIER ::= SPECIAL_CHARACTER { SPECIAL_CHARACTER } ;
```

3.7 Keywords

The identifiers given in table 3.1 are keywords of Lotos NT. They are used for syntactic purpose and cannot be used as normal identifiers. They are written between double quotes in the concrete syntax and in boldface in the abstract syntax.

and	andthen	any	array	as	behaviour
block	by	case	choice	do	else
elsif	end	enum	eqns	eval	exception
exceptions	exit	external	false	finite	for
forall	function	gate	gates	generic	hide
if	iff	implies	import	in	inout
interface	is	library	list	loop	module
not	null	of	ofsort	opns	or
orelse	out	par	process	procs	raise
raises	record	rename	renaming	return	set
signal	specification	stop	then	trap	true
type	types	value	var	xor	wait
when	while	with			

Table 3.1: The keywords of Lotos NT

3.8 Literals

A literal is the source code representation of a value of a primitive type (§ 4.3, p. 29).

```
Literal ::= INTEGER
          | FLOATING_POINT
          | CHAR
          | STRING ;
```

3.8.1 Integer Literals

See Section 4.3.3 for a general discussion of the integer types and values.

Integer literals may be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8):

```
INTEGER ::= DECIMAL_NUMBER
          | HEX_NUMERAL
          | OCTAL_NUMERAL ;
```

A decimal numeral is either the single ASCII character 0, representing the integer zero, or consists of an ASCII digit from 1 to 9, optionally followed by one or more ASCII digits from 0 to 9, and represents a positive integer.

```
DECIMAL_NUMBER  ::= "0"
                |   NON_ZERO_DIGIT { "_" | DIGIT } ;

NON_ZERO_DIGIT  ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

The simplest form of integer literal is simply a sequence of decimal digits. If the literal is very long, it may be convenient to split it up into groups of digits by inserting isolated underlines thus "`123_456_789`". In contrast with identifiers, such underlines, are of course, of no significance other than to make the literal easier to read.

NOTE: The use of "_" character to format integers is also adopted by ADA language.

A hexadecimal numeral consists of the leading ASCII characters 0x or 0X followed by one or more ASCII hexadecimal digits and can represent a positive, zero, or negative integer. Hexadecimal digits with values 10 through 15 are represented by the ASCII letters a through f or A through F, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

```
HEX_NUMBER      ::= "0" ("X" | "x") HEX_DIGIT { HEX_DIGIT } ;

HEX_DIGIT       ::= "0".."9" | "a".."f" | "A".."F" ;
```

An octal numeral consists of an ASCII digit 0 followed by one or more of the ASCII digits 0 through 7 and can represent a positive, zero, or negative integer.

```
OCTAL_NUMBER    ::= "0" OCTAL_DIGIT { OCTAL_DIGIT } ;

OCTAL_DIGIT     ::= "0".."7" ;
```

Lexically correct integers may be refused by *compilers* if they denote values which do not fit (implementation dependent) range of type `int` (or `nat`).

3.8.2 Floating-Point Literals

See Section 4.3.4 for a general discussion of the floating-point types and values.

A floating-point literal has the following parts: a whole-number part, a decimal point (represented by an ASCII period character), a fractional part, and an exponent. The exponent, if present, is indicated by the ASCII letter e or E followed by an optionally signed integer.

At least one digit, in either the whole number or the fraction part, and a decimal point are required. The exponent is optional.

```
FLOATING_POINT ::= DECIMAL_NUMBER "." [ DIGIT { ["_"] | DIGIT } ]
                   [ ( "e" | "E" ) [ "+" | "-" ] DECIMAL_NUMBER ]
                 | DECIMAL_NUMBER ( "e" | "E" ) [ "+" | "-" ] DECIMAL_NUMBER
                 | "." DIGIT { ["_"] | DIGIT }
                   [ ( "e" | "E" ) [ "+" | "-" ] DECIMAL_NUMBER ];
```

Lexically correct floating point numbers may be refused by *compilers* if they denotes values which do not fit (implementation dependent) range of type `float`.

Examples of floating-point literals:

```
1e1      2.       .3      0.0      3.14      6.022137e+23      1e-9
```

3.8.3 Characters

A character literal is expressed as a character or an escape sequence, enclosed in ASCII single quotes. A character literal is always of type `char`. See Section 4.3.5 for more details on the `char` type.

```
CHAR ::= "'" CHAR_PRINTABLE "'" ;

CHAR_PRINTABLE = PRINTABLE | "\"" ;

PRINTABLE ::= TRULY_PRINTABLE
            | "\n"     -- linefeed LF
            | "\t"     -- horizontal tab HT
            | "\v"     -- vertical tab VT
            | "\b"     -- backspace BS
            | "\r"     -- carriage return CR
            | "\f"     -- form feed FF
            | "\a"     -- alert BEL
            | "\\"     -- backslash
            | "\?"     -- question mark
            | "\'"     -- single quote '
            | "\""     -- double quote "
            | "\\" OCTAL_DIGIT [ OCTAL_DIGIT [ OCTAL_DIGIT ]]
            | "\\" "x" HEX_DIGIT [ HEX_DIGIT ] ;

TRULY_PRINTABLE = CHARACTER - "\'\"\\" ;  -- printable characters

CHARACTER       = #040..#176 + #240..#377 ;
```

The escape sequences allow for the representation of some non graphic characters as well as the single quote, double quote, query, and backslash characters in character literals and string literals.

It is a compile-time error for the character following the `TRULY_PRINTABLE` or `ESCAPE_SEQUENCE` to be other than a '.

It is a compile-time error for a line terminator to appear after the opening ' and before the closing '.

It is a compile-time error if the character following a backslash in an escape is not from the set specified above.

The following are examples of char literals:

```
'a'    '%'    '\t'    '\\'    '\''    '\xFFFF'    '\177'
```

3.8.4 String Literals

A string literal consists of zero or more characters enclosed in double quotes. Each character may be represented by an escape sequence.

A string literal is always of type `string` (§ 4.3.6, p. 32).

```
STRING ::= "\"" { STRING_PRINTABLE } "\"" ;

STRING_PRINTABLE = PRINTABLE | "\'" ;
```

As specified in Section (§ 3.2, p. 20), neither of the characters CR and LF is ever considered to be `PRINTABLE`; each is recognized as constituting a line terminator. Instead, one should use the escape sequences "`\n`" for LF and "`\r`" for CR.

It is a compile-time error for a line terminator to appear after the opening " and before the closing matching ".

The following are examples of string literals:

```
""    "\""    "\n"   "This is a string"
```

3.9 Operators

The following special symbols are reserved tokens of the languages, formed from ASCII characters. They appear into the concrete syntax between double quotes and in the abstract syntax in teletype fonts.

```
Operator ::= "->" | "@"  | "}"   | "]"  | ")"  | ","  | ":"  | "::"
         |   ":=" | "[>" | "..." | "="  | "==" | "!"  | "[]" | "=>"
         |   ">=" | ">"  | "|||" | "|[" | "<=" | "<"  | "|"  | "-"
         |   "%"  | "!=" | "#"   | "{"  | "["  | "("  | "+"  | "**"
         |   "?"  | "]|" | ";"   | "*" ;
```

Chapter 4

Types

Lotos NT is a strongly typed language, a necessary condition for ensuring description safety.

Type declaration are used to define new types when the few predefined types are insufficient, which is the case of most descriptions. The declaration of new types is very general. However, several well-know type schemes[1] may be derived. In this case, some implicit declarations of other Lotos NT objects appear.

4.1 Type Definition

A type denotes a domain of values (see Section 2.4) on which Lotos NT objects are defined.

In Lotos NT, a type definition must be associated with a name which will be used to refer to it where useful: this association is a type declaration. This means that anonymous types do not exist. For this reason, the equality between types is given by the equality of their names (instead of structural equality).

The definition of types in Lotos NT follows the general approach of constructed types in functional languages where types are defined using type *constructors*. Constructors are special operations structuring the domain of the type. They give a name to the sub-domain of the type represented by the Cartesian product of the parameters.

The simpler syntax for type definition is the following:

$$
\begin{array}{l}
\textbf{type } T \textbf{ is} \\
\quad C_1 \ [(V_1^1 : T_1^1, ..., V_{m_1}^1 : T_{m_1}^1)] \textbf{,} \\
\quad \cdots \textbf{,} \\
\quad C_n \ [(V_1^n : T_1^n, ..., V_{m_n}^n : T_{m_n}^n)] \\
\textbf{end type}
\end{array}
$$

where $T, T_1^1, ...$ are type identifiers, $C_1, ..., C_n$ are constructor identifiers, and $V_1^1, ...$ are variable identifiers. The default list of parameters is the empty list.

For a constructor C_i, the identifier V_j^i is called *field* or formal parameter, and m_i is called operation arity.

The syntax given above must satisfy the following static semantics constraints:

[1]For Lotos, these schemes are also known as "rich term syntax" [Pec94].

- There must be at least one constructor declaration ($n \geq 1$).
- For a given constructor C_i, the names of formal parameters must be pairwise distinct, *i.e.*, $\forall j, k \in \{1, ..., m_i\} \quad (j \neq k) \Longrightarrow (V^i_j \neq V^i_k)$.
- For the set of constructors of a given type, fields having the same name should have the same type. For example, the type `HeaderType` defines the values that a header may have:

  ```
  type HeaderType is
      Header1 (dest_id, data_length, header_CRC: nat)
  |   Header2 (dest_id: nat, source_id: nat, data_length, header_CRC: nat)
  entype
  ```

  The field `dest_id` appears in the parameter list of the two constructors with the same type `nat`. Note that the fields having the same type may be grouped in lists, like for `Header1` constructor.
- Two or more constructors may have the same name (may be overloaded) if their profiles (the list of the *types* of parameters and the result type) differ. Note that the name of formal parameters does not solve the overloading.
- Type declarations may be mutually recursive. However, each type must be *productive*, *i.e.*, it must have at least one value. Formally, a type is productive iff: (a) it has a constructor of arity 0 or (b) all the parameters of its constructors have productive types.

Example 4.1.1
The type “`bool`” is defined in the (predefined) standard library as an enumeration of two values **true** and **false**, which are the type constructors of arity 0.

```
type bool is
    false,
    true
end type
```

A more elaborate type is the type of a packet which contains a header part and a data part:

```
type PacketType is
    Packet (header: HeaderType, data: DataType)
end type
```

The constructor of type `PacketType`, `Packet`, has two parameters: the first is named `header` and has the type `HeaderType`, the second is named `data` and has the type `DataType`.

A list of packets may be defined using a recursive definition:

```
type PacketListType is
    PacketList_empty,
    PacketList_cons (head: PacketType, tail: PacketListType)
end type
```

The lists may be defined also using the rich term syntax as described in Section 4.4. ■

4.2 Predefined Operations

For each definition of a constructed type T, a set of predefined operations are automatically generated (*sheel* definitions of [BM79]):

- "`==`" and "`!=`", with the profile $T, T \rightarrow$ `bool`, for the equality (*resp.* non equality) test.
- "`<`", "`>`", "`<=`", "`>=`", with the profile $T, T \rightarrow$ `bool`, for the ordering test of values. Note that values of constructed types are ordered lexicographically. The declaration order of constructors is important: the constructor declared first is less than the constructors following it in the declarations.
- "`string`", with the profile $T \rightarrow$ `string`, returns the string representation of the value given as parameter.
- "`ord`", with the profile $T \rightarrow$ `nat`, returns the order number of the (first) constructor of the value.
- "`card`", with the profile $T \rightarrow$ `nat`, returns the number of constructors for T.

Only for finite types (§ 4.4.2, p. 34), the following operations are also defined :

- "`succ`" and "`pred`", with the profile $T \rightarrow T$, return the successor (*resp.* the predecessor) of the value given as parameter. For the border values, these operation are identities.
- "`hash`", with the profile $T \rightarrow$ `nat`, returns the order number of the term in the domain of the type T.

The user may specify explicitly the operations to be automatically generated when the type is declared, using a "**with**" clause:

```
type T is
  ...
  [with op1, ..., opn]
end type
```

where $op_1, ..., op_n$ belong to the set of the predefined operations above. Note that "`==`" and "`!=`" are always generated.

4.3 Predefined Types

As stated in the introduction, a "pure" FDT should not make assumption about the implementation issues. The FDT LOTOS respects this constraint by allowing for types like natural numbers or integers only an axiomatic definition. In order to make easier the user task, the standard provides a standard library of data types which contains types like: boolean, natural number, bit, octet, etc.

However, feedback from users showed that the axiomatic definition is not natural and easy to use (e.g., natural numbers where 13 is expressed by 13 compositions of the operation "Succ" applied to "0"!). By consequence, it seems useful to accept natural (programming languages) notations for a set of predefined types. Chapter 3 defines the lexical tokens corresponding to these constants. This alternative definition does not exclude implementation dependent definitions given by the compilers[2].

This section presents some of the predefined types which form the static basis of any LOTOS NT description.

[2]For example, TRAIAN provides such an implementation in the file `incl/lotosnt_predefined.h`.

4.3.1 The boolean type

Values of the boolean type, written "`bool`", are usual truth values **true** and **false**.

Besides the predefined operations provided for usual types, additional operations are available on type `bool`, e.g., the binary conjunction and disjunction, the unary negation, and comparisons (**false** < **true**). Binary operations may exist in strict and non-strict (short-circuit evaluation) versions. An exhaustive list of these operations is given in Table 4.1.

Name	Profile	Description
`not`	`bool` → `bool`	boolean negation
`or`	`bool`, `bool` → `bool`	logical disjunction
`orelse`	`bool`, `bool` → `bool`	cancellative or
`and`	`bool`, `bool` → `bool`	logical conjunction
`andthen`	`bool`, `bool` → `bool`	cancellative and
`implies`	`bool`, `bool` → `bool`	logical implication
`iff`	`bool`, `bool` → `bool`	logical equivalence
`xor`	`bool`, `bool` → `bool`	exclusive or

Table 4.1: Predefined operations on type `bool`

4.3.2 The natural type

Values of natural type, written "`nat`", are natural numbers.

Besides the predefined operations provided for usual types, additional operations available on type `nat` are, for instance, binary operations such as addition, subtraction, multiplication, (Euclidean) quotient and remainder, and conversions to other numerical types. An exhaustive list of these operations is given on Table 4.2.

Name	Profile	May raise	Description
`+`	`nat`, `nat` → `nat`		addition
`-`	`nat`, `nat` → `nat`	`RANGE_ERROR`	subtraction
`*`	`nat`, `nat` → `nat`		multiplication
`**`	`nat`, `nat` → `nat`		power
`/`	`nat`, `nat` → `nat`	`ZERO_DIVISION`	division
`%`	`nat`, `nat` → `nat`	`ZERO_DIVISION`	remainder (modulus)
`int`	`nat` → `int`	`RANGE_ERROR`	integer conversion
`char`	`nat` → `char`	`RANGE_ERROR`	char conversion
`real`	`nat` → `real`		real conversion

Table 4.2: Predefined operations on type "`nat`"

4.3.3 The integral type

Values of integral type, written "`int`", are signed naturals.

Besides the predefined operations provided for usual types, additional operations available on type `int` are, for instance, binary operations such as addition, subtraction, multiplication, (Euclidean) division, sign inversion, and conversions to other numerical types. An exhaustive list of these operations is given on Table 4.3.

Name	Profile	May raise	Description
`sign`	`int` → `int`		sign
`-`	`int` → `int`		sign inversion
`+`	`int`, `int` → `int`		addition
`-`	`int`, `int` → `int`		subtraction
`*`	`int`, `int` → `int`		multiplication
`**`	`int`, `nat` → `int`		power
`/`	`int`, `int` → `int`	`ZERO_DIVISION`	division
`%`	`int`, `int` → `int`	`ZERO_DIVISION`	remainder (modulus)
`abs`	`int` → `int`		absolute value
`nat`	`int` → `nat`	`RANGE_ERROR`	natural conversion
`char`	`int` → `char`	`RANGE_ERROR`	char conversion
`real`	`int` → `real`		float conversion

Table 4.3: Predefined operations on type `int`

4.3.4 The floating point type

Values of the floating point type, written “`real`”, are signed floating point numbers. Tools may consider implementation defined approximations of real numbers in an implementation-defined range.

Besides the predefined operations provided for usual types, additional operations on these values are the usual arithmetic operations, and conversions to another type. An exhaustive list of these operations is given in Table 4.4.

Name	Profile	May raise	Description
`-`	`real` → `real`		sign inversion
`abs`	`real` → `real`		absolute value
`+`	`real`, `real` → `real`		addition
`-`	`real`, `real` → `real`		subtraction
`*`	`real`, `real` → `real`		multiplication
`/`	`real`, `real` → `real`	`ZERO_DIVISION`	division
`**`	`real`, `nat` → `real`		power
`int`	`real` → `int`	`RANGE_ERROR`	int conversion

Table 4.4: Predefined operations on type `real`

4.3.5 The character type

Values of character type, written “`char`”, are characters of the ASCII alphabet. Additional operations available on these values are, e.g., conversion into other types. An exhaustive list of these operations is given in Table 4.5.

NOTE: Different character sets may be considered. Here, we consider for the predefined character type the ISO Latin-1 character set.

Name	Profile	May raise	Description
`nat`	`char` → `nat`		natural conversion
`tolower`, `toupper`	`char` → `char`		conversion
`isupper`, `islower`, `isalpha`, `isdigit`, `isxdigit`, `isalnum`	`char` → `bool`		tests

Table 4.5: Predefined operations on type `char`

4.3.6 The string type

Values of string type, noted "`string`", are dynamic-length character strings.

Additional operations available on these values may be concatenation, getting length of a string, taking a substring of a longer string, taking all of a string except for a substring, inserting a string into another one, searching a given substring in a longer string ... Strings are ordered by lexicographic order. Strings may also be converted to other types. An exhaustive list of these operations is given in Table 4.6.

Name	Profile	May raise	Description
`length`	`string` → `nat`		length
`concat`	`string`, `string` → `string`		concatenation
`index`, `rindex`	`string`, `string` → `nat`		sub-string search
`prefix`, `suffix`	`string`, `nat` → `string`		sub-string selection
`substr`	`string`, `nat`, `nat` → `string`		sub-string selection
`nth`	`string`, `nat` → `char`		the n-th character
`empty`	`string` → `bool`		emptiness test
`int`	`string` → `int`	`RANGE_ERROR`	int conversion
`real`	`string` → `real`	`RANGE_ERROR`	real conversion

Table 4.6: Predefined operations on type `string`

4.3.7 The "none" type

The type "`none`" is the type whose domain is empty.

The type `none` is useful to give an uniform syntax for exceptions without value parameter, for functions without result (§ 5.5.1, p. 54), for non-exiting processes (§ 6.7.1, p. 66), and for typing expressions and statements.

4.3.8 The "time" type

The case of type "`time`" is special in LOTOS NT. It denotes values of the time domain. In some languages like: ATP[NS94], ...the time domain must be discrete, *i.e.*, isomorphic to the natural numbers. In LOTOS NT (as introduced by its predecessors ET-LOTOS or RT-LOTOS), the time domain may be dense[3], *i.e.*, isomorphic to non-negative rational numbers.

The unique assumption made for the time domain is that it is countable. This implies that the underlying semantics model is indeed a labelled transition system (§ 2.5, p. 17). In fact, it is possible

[3]An ordered set is dense if it is possible to find a value between any two different given values.

with Lotos NT to define one's own time domain, noted D, provided that some assumptions are satisfied:

A1 There exists an associative and commutative operation "$+ \ : \ D, D \to D$".

A2 There exists an element $0 \in D$ such that it is neutral element for "+".

A3 The domain D is left-cancellative, *i.e.*, $\forall t, u \in D \ . \ (t + u = t + v) \Longrightarrow (u = v)$.

A4 The domain D is anti-symmetric, *i.e.*, $\forall t, u \in D \ . \ (t + u = 0) \Longrightarrow (t = u = 0)$.

A5 Each time domain $(D, +, 0)$ induces a binary order of *precedence* defined by:

$$\forall t, u \in D \ . \ t \leq u \Longleftrightarrow \exists v \in D \ . \ t + v = u$$

A6 There exists an absorption element ∞, defined such that $\forall t \in D \ . \ t + \infty = \infty$. So $\forall t \in D \ . \ t \leq \infty$.

The assumptions **A1**–**A4** imply that the time domain is a left-cancellative anti-symmetric monoid [JSV93]. Examples of time domains include: the natural numbers ($\texttt{nat}^+ \cup \{\infty\}, +, 0$), the positive rational numbers, etc.

NOTE: In the current version, Traian considers that the time domain is `nat`.

4.4 Derived Types

The Lotos NT type declaration is very general. It allows the definition of more familiar types (e.g., enumerations, records, sets, lists, arrays) as derived types or "rich term syntax" [Pec94].

This section presents how some derived type declarations are introduced as syntactic sugar of the more general type declaration.

4.4.1 Enumerated types

The *enumerated* type definition declares values of a (finite) domain ordered by the declaration order. The declaration of an enumerated type T_E with values C_1, ..., C_n has the following syntax:

type T_E **is**
  enum C_1 **,** ... **,** C_n
  [**with** $op_1, ..., op_n$]
end type

For example:

```
type day_of_week is
    enum Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
    with <
end type
```

The order in which the values of the type are declared induces an order relation, for example `Monday < Wednesday`.

The declaration above is translated into the following constructed type declaration:

type T_E **is**
  C_1 **,** ... **,** C_2
  [**with** $op_1, ..., op_n$]
end type

In Table 4.7 we give the exhaustive list of the additional operations for an enumerated type T_E.

Name	Profile	May raise	Description
`min`†	$\rightarrow T_E$		least value
`max`†	$\rightarrow T_E$		greatest value
T_E†	`nat` $\rightarrow T_E$	`RANGE_ERROR`	T_E conversion

Table 4.7: Operations predefined for enumerated types

4.4.2 Scalar and simple types

Scalar types are `bool`, `nat`, `int`, `char`, user defined *finite* types and *enumerable* types, and those only.

A type is *finite* if its domain is finite. Informally, a type is finite if either its constructors are of arity 0 (enumerated types), or the arguments of its constructors are finite types which do not depend recursively on the current type.

Formally, it is possible to detect finite types by constructing the dependency graph between types. A type T depends on type T' if T' appears as the type of an argument of a constructor of type T. The finite types are those contained in acyclic sub-graphs (trees) of the dependency graph having as leaves user defined enumerated types, or `bool`, or `char`.

For finite types T it is possible to define:

- a function `init`† $: \rightarrow T$ which gives the smallest value of the domain,
- a function `succ`† $: T \rightarrow T$ which gives for a value the "next" value of the domain, and
- a function `max`† $: \rightarrow T$ which gives the biggest value of the domain.

An *enumerable* type is a type whose domain is isomorphic with the domain of natural numbers, and a total order relation defined on its elements. For the user defined types, the order relation is given by the lexicographic order induced by the declaration order of the constructors. Consider for example the type `HeaderType` (§ 4.1, p. 27). Values constructed using the `Header1` constructor are smaller than values constructed using the constructor `Header2`.

For enumerable types, it is possible to define the functions `init` and `succ`, but not `max`.

Simple types are scalar types plus the type `real`.

4.4.3 Record types

A *record* type corresponds to the Cartesian product of component types, except that component values are accessed by a name rather than by their position.

A simple syntax for record type definition T_R with fields V_1 of type T_1, ..., V_n of type T_n is:

```
type $T_R$ is
   record $V_1 : T_1, ..., V_n : T_n$
   [with $op_1, ..., op_n$]
end type
```

The declaration above is translated into the following one:

> **type** T_R **is**
> T_R(${V_1}$:T_1, ..., V_n:T_n)
>   [**with** $op_1, ..., op_n$]
> **end type**

and operations as selection of a field, equality, inequality, and comparisons are defined as for the constructed types.

4.4.4 Sets

A *set* type declaration introduces a type which is the power-set of some other *scalar* type, which is called the *basic* type. The sets cannot have multiple occurrences of the same element. In order to test if an element is equal to another, an equality relation should be defined over the domain of the basic type.

The definition of a set type T_{set} with elements of the scalar type T has the following syntax:

> **type** T_{set} **is**
>   **set of** T
>   [**with** $op_1, ..., op_n$]
> **end type**

Values of such types are subsets of the set of all values of the basic type. Any set, whatever its size is, can be represented, except when system-dependent limitation arises. Implementations may use some techniques to be space and time efficient, but they cannot apply a general technique. For example, a possible representation of a set type T_{set} with elements of type T is given by the following constructed type definition:

> **type** T_{set} **is**
>   `NIL , CONS(HEAD:`T`,TAIL:`T_{set}`)`
>   [**with** $op_1, ..., op_n$]
> **end type**

where `NIL` denotes the empty set, and `CONS` denote the union between an element `HEAD` and a set `TAIL`. Note that the union is not disjunct, so `HEAD` can be an element of `TAIL`.

Operations available on sets are construction, binary operations as union, intersection, and difference, comparisons, and membership test. An exhaustive list of these operations for a type T_{set} which is a set with elements of a given type T is presented in Table 4.8.

The values of type set may be represented "in extenso" (*i.e.*, by giving the list of their elements) using the notation:

> `{`$E_1, ..., E_n$`}`

where E_1, ..., E_n are expressions returning values of type T.

This notation must be equivalent to the value `union(CONS(`E_1`,NIL),...)` in the representation given above.

4.4.5 Lists

Values of type *list* are ordered, linear lists, the elements of which belong to the same type, called the *element* type. There is no restriction on this type.

Name	Profile	May raise	Description
`{}`†	$\rightarrow T_{set}$		empty set
`full`†	$\rightarrow T_{set}$		full set
`union`†	$T_{set}, T_{set} \rightarrow T_{set}$		union
`diff`†	$T_{set}, T_{set} \rightarrow T_{set}$		difference
`comp`†	$T_{set} \rightarrow T_{set}$		complementation
`inters`†	$T_{set}, T_{set} \rightarrow T_{set}$		intersection
`isin`†	$T, T_{set} \rightarrow$ `bool`		membership test
`isempty`†	$T_{set} \rightarrow$ `bool`		emptiness test
`issubset`†	$T_{set}, T_{set} \rightarrow$ `bool`		emptiness test
`isdisjoint`†	$T_{set}, T_{set} \rightarrow$ `bool`		emptiness test
`card`†	$T_{set} \rightarrow$ `int`	`RANGE_ERROR`	number of elements

Table 4.8: Operations predefined for a set type

The definition of a list type T_L with elements of type T has the following syntax:

type T_L **is**
 list of T
 [**with** $op_1, ..., op_n$]
end type

This definition is translated into a constructed type definition as follows:

type T_L **is**
 `NIL, CONS(HEAD:`T`,TAIL:`T_L`)`
 [**with** $op_1, ..., op_n$]
end type

Operations available for lists are construction, efficient non-destructive concatenation, length, test for emptiness, *head*, and *tail*. An exhaustive list of these operations for a type T_L (list of elements of type T) is presented in Table 4.9.

Name	Profile	May raise	Description
`nil`	$\rightarrow T_L$		empty list
`cons`	$T, T_L \rightarrow T_L$		list cons (front)
`isempty`†	$T_L \rightarrow$ `bool`		emptiness test
`tcons`†	$T_L, T \rightarrow T_L$		list cons (end)
`head`†	$T_L \rightarrow T$	`EMPTY_LIST`	first element
`tail`†	$T_L \rightarrow T_L$	`EMPTY_LIST`	all elements but first
`concat`†	$T_L, T_L \rightarrow T_L$		concatenation
`length`†	$T_L \rightarrow$ `int`	`RANGE_ERROR`	number of elements
`nth`†	T_L, `nat` $\rightarrow$ T	`RANGE_ERROR`	n-th element

Table 4.9: Operations predefined for a list type

The list may be specified "in extenso" by using the following notation:

`[`$E_1, ..., E_n$`]`

where E_1, ..., E_n are expressions returning values of type T. This notation is equivalent to `CONS(`E_1`,CONS(`E_2`,...,CONS(`E_n`,NIL)))`.

4.5 Subtypes†

A *subtype* type gives a name of a sub-domain of a base type, named *host type*, by specifying the set of values of this sub-domain by a boolean predicate. The syntax for subtype definition T_s of host type T is:

> **type** T_s **is**
> $\{V : T,\ E\}$
> [**with** $op_1, ..., op_n$]
> **end type**

where E is a predicate on the type T. A subtype is a new type in the sense of the type checking: values of a subtype type are considered different from the values of the host type. The conversion between values of the subtype and value of the host type should be done explicitly using "**of**" expressions. The operations in the subtype types are those inherited from the host type.

4.6 Arrays†

Values of type *array* are tuples of a fixed dimension, the element of which belong to the same type, called the *base* type. Base types must be finite so that arrays are finite. The dimension of the array should be finite.

The definition of an array type T_A of dimension n, with indexes from the product of types $T_1, ..., T_n$, and elements of type T_0 has the following syntax:

> **type** T_A **is**
> **array** $[T_1, ..., T_n]$ **of** T_0
> **end type**

4.7 External Types and Pragmas

In order to interface with other languages, a type definition may specify the name which should be used for the type in an implementation. Moreover, it can also specify that a type has an external definition. This is done using *pragmas*.

The general syntax for type definition becomes:

> **type** T **is** *type-pragmas*
> C_1 $[(\overline{V}^1_1 : T^1_1, ..., \overline{V}^1_{m_1} : T^1_{m_1})]$ *operation-pragmas* **,**
> $\cdots$ **,**
> C_n $[(\overline{V}^n_1 : T^n_1, ..., \overline{V}^n_{m_n} : T^n_{m_n})]$ *operation-pragmas*
> [**with** $op_1, ..., op_n$]
> **end type**

Types pragmas are lists of *type-pragma* having the following forms:

- "`!external`" if the type has an external implementation; this implementation should be provided in an external file having the extension "`.t`".
- "`!implementedby STRING`" if the external name used for the type is `STRING`.
- "`!comparedby STRING`" if the equality function `==` should be implemented by the `STRING` function.

- "`!printedby STRING`" if the values of the type should be printed by the `STRING` function.
- "`!enumeratedby STRING`" if the values of the type should be enumerated (if possible) by the `STRING` function.
- "`!pointer [INTEGER]`" if the type has to be implemented by a pointer in C.
  When `INTEGER` is strictly positive, then all values of type T will be stored into an hash table of size `INTEGER` and, thus, represented as entries within this table. The hash table is extensible, meaning that T can have more than `INTEGER` different values. For performance reasons, it is advised to choose a value of `INTEGER` close to the cardinal of T.
  When `INTEGER` is not specified, the type will be implemented as a pointer type, instead of having its value stored in an hash table.

Operation pragmas are lists of *operation-pragma* having the following forms:

- "`!external`" if the operation has an external implementation in a file having the extension "`.f`".
- "`!implementedby STRING`" if the external name of the operation is `STRING`.

Chapter 5

Expressions, Statements, and Functions

The data part of LOTOS NT is mainly based on types, expressions, statement, and functions. Types and type definitions are presented in Chapter 4; expressions, statements, and functions are presented in this chapter.

One may replace the data part of LOTOS NT with another data description formalism ensuring the safety property, for example ACTONE data types. Note that the behaviour part of the language contains symmetrical constructs of the data part.

There is a fundamental difference between the expressions/statements (and functions) and behaviours (and processes). An expression or a statement cannot contain communication, non-determinism, concurrency, or real-time. The data part is a sequential and deterministic language. The evaluation of an expression takes no time, and should return a value, or may raise an exception. The evaluation of a statement takes no time, and may return a value, assign some variables, and raise an exception.

An important characteristic of the data language presented here is its "clean" imperative style. The language supports assignment and other imperative style facilities, but a proper semantics is given [Sig99], which restricts undesirable effects like the use of uninitialized variables. The imperative style is combined with constructs specific of functional languages, e.g., *pattern-matching*.

This chapter presents all aspects related to data language: constants, variables, expressions, statements, and function declarations.

5.1 Constants

Primitive constants, written K, are boolean values **true** and **false**, integral numbers of type `int` or `nat`, floating point numbers of type `real`, character constants of type `char`, and string constants of type `string`.

5.2 Value expressions

Value expressions (or shortly expressions) are primitive constants and terms of the user defined types built using the application of constructors or functions. As a consequence, any value is typed. Moreover, value expressions cannot assign variables and may raise exceptions (§ 5.4.9, p. 52) only by function calls.

The following grammar gives the syntax of value expressions:

E	$::=$	K	*primitive constant*	(E1)
	$\mid$	V	*variable*	(E2)
	$\mid$	$C\ [(ES)]$	*constructor application*	(E3)
	$\mid$	$E\ C\ E$	*infix constructor application*	(E4)
	$\mid$	$[\ E\ \{,E\}\ ]$	*list or set construction*	(E5)
	$\mid$	**array** $T\ [\ E\ \{,E\}\ ]$	*array construction*	(E6)
	$\mid$	$F\ [(ES)]$	*function call*	(E7)
	$\mid$	$E\ F\ E$	*infix function call*	(E8)
	$\mid$	$V\ [E_1, ..., En]$	*array access*	(E9)
	$\mid$	$E.V$	*field selection*	(E10)
	$\mid$	$E.'\{'\ ES\ '\}'$	*field update*	(E11)
	$\mid$	E **of** T	*type coercion*	(E12)
	$\mid$	E **raises** $[XS]$	*raise exception expression*	(E13)
	$\mid$	(E)	*parenthesized expression*	(E14)

ES	$::=$		*empty record*	(ES1)
	$\mid$	V_1=>$E_1, ..., V_n$=>E_n	*record*	(ES2)
	$\mid$	$E_1, .., E_n$	*tuple*	(ES3)

XS	$::=$	$[\ldots]$	*empty or dotted record*	(XS1)
	$\mid$	X'_1=>$X_1, ..., X'_n$=>$X_n\ [,\ldots]$	*record*	(XS2)
	$\mid$	$X_1, .., X_n$	*tuple*	(XS3)

where $E, E_1, ...$ denote value expressions, V denotes variables, C denotes constructor identifiers, ES denotes lists of actual value expression parameters (named for record or unnamed for tuple), and XS is a list of actual exception parameters (§ 5.4.9, p. 52). The precedence of operators appearing in expressions is given in table 5.2.

Priority	*Operations*
0.	**raises**
1.	`-` unary
2.	**of**, `.` field selection and update
3.	`and`, `andthen`, `/`, `%`, `*`, `**`
4.	`or`, `orelse`, `xor`, `+`, `-`
5.	`==`, `!=`, `<`, `<=`, `>=`, `>`, `iff`, `implies`

Value expressions are evaluated to *value* or normal forms. Values are primitive constants and ground terms of the user defined types built using the application of constructors on value records. We will write values $N, N_1, ...$ and sequences of values NS.

The construct "`...`" for exception parameters allows the user to avoid the explicit instantiation of the actual exception parameters if they are the same name as the formal parameters. For example, if the formal parameter exception of the function F is called X, one may call F like "F `()` **raises** `[...]`" which is replaced by the compiler with "F `()` **raises** `[`X`=>`X`]`"; in this case, X should be already declared in the environment.

5.2.1 Variables

Variables, noted V, are *assignable* objects containing values which are computed elsewhere. Note that, from this point of view, the LOTOS NT data language is an imperative language: it supposes the existence of a *memory* (a set of cells represented by variables which can store some values) which can be accessed for read and write operations. However, the static semantics constraints impose a clean imperative style: the access to an uninitialized cell (variable) is signaled at compile time and does not produce a run-time error.

A value expression may be a variable V. The variable must be initialized (contains a value). The type of the expression is the type of the variable, and the result of the expression evaluation is the value of the variable V.

Chapter 6 presents the use of variables in behaviours. The main constraint is that variables cannot be shared between concurrent behaviours. More precisely, if a variable is written by a behaviour, it cannot be read or written by a concurrent behaviour.

5.2.2 Constructor application

The constructor application computes values of the domain of their target type. The constructor should be already defined in the current environment by type definitions (§ 4.1, p. 27). The actual list of arguments of the constructor may be expressed either *by name* giving a record whose fields are labelled with the names of formal parameters (alternative ES2), or *by position* giving the (ordered) list of actual values (alternative ES3).

The expressions below use positional constructor application:

C `(`E_1`, ..., `E_n`)`
E_1 C E_2

the following expression use the named style:

C `(`V_1`=>`E_1`, ..., `V_n`=>`E_n`)`

Note that the positional style may be translated into the named style using the static semantics informations about the constructor declaration.

The number of actual parameters must be the same as the arity of the constructor. In the named style, the names of the formal parameters (fields $V_1, ..., V_n$) must be pairwise distinct, *i.e.*, a formal parameter V_i should appear only once in the list. The values $E_1, ..., E_n$ associated with these names must have the same types as the corresponding formal parameters.

If the constructor has only two parameters it can be applied in the infix positional style.

If the constructor is overloaded, the informations given by the type of its parameters and the type of the resulting value should suffice to solve the overloading (*i.e.*, to find the unique constructor having this profile).

The evaluation of constructor application begins with the left-to-right evaluation of its actual parameters. The values obtained are used to form the constructed value which is the result of evaluation. If one expression E_i raises an exception (§ 5.4.9, p. 52), the evaluation is blocked, and the exception is signaled.

Example 5.2.1
`Monday` is a value of the type `day-of-week`.

`Header1 (1, 2, 3)`, `Header2 (1, 0, 2, 3)` are values of type `HeaderType`.

`Header1 (1, 2, 3)` and `Header1 (data_length => 2, dest_id => 1, header_CRC => 3)` represent the same value of type `HeaderType`. ■

5.2.3 Function application

The function application is largely treated in Section 5.5.2. Note that, for function application expressions the function should return a value and cannot do side effects (have only "**in**" parameters).

Functions may be predefined operations described in 4.3.

5.2.4 Field selection

A field selection expression has the general form:

(E.V) **raises** [X]

where E is an expression and its value is of the form C`(`V_1`=>`$N_1, ..., V_n$`=>`N_n`)` (*i.e.*, a constructed value) and V is one of the fields $V_1, ..., V_n$. The selection expression returns the value of this field. Note that the parenthesis are needed in order to avoid some ambiguities in the grammar.

It is worth noticing that the exception X is raised (see Section 5.4.9) if the value returned by E does not have a field of name V. In fact, the static semantics ensures that the field V is a field of one of the constructors of the E type. However, this does not suffice to ensure that no dynamic error arises.

NOTE: To be sure that no exception is raised, one may wish to use field selections for record types only. In this case, the static semantics will ensure that the single constructor of the record type has the field as argument; thus, the raise clause should be omitted with current version of the compiler.

Example 5.2.2
If E is an expression of type `HeaderType`, the following expression:

(E`.dest_id`) **raises** [X]

selects the component "`dest_id`" of the value. More precisely, if E evaluates to "`Header1 (data_length => 2, dest_id => 1, header_CRC => 3)`", the selection expression above returns 2; if E evaluates to "`Header2 (1, 0, 2, 3)`", the selection expression evaluates to 1 (the field "`dest_id`" is the first parameter of the "`Header2`" constructor). ■

The field selection may be translated into a function call. For example, if E is an expression of type T below:

```
type T is
    C_1(V_1 : T_1)
  | C_2(V_2 : T_2)
  | C_3(V_1 : T_1, V_2 : T_2)
  | C_4(V_4 : T_4)
end type
```

then, the expression `(`E`.`V_1`)` **raises** `[`X`]` is equivalent to a function call having the following body:

```
case E is
    C_1(V_1 : T_1)
  | C_3(V_1 : T_1, V_2 : T_2) -> return V_1
  | any T -> raise X
end case
```

5.2.5 Field update

A field update expression has the general form:

`(`E`.{`V_1`=>`E_1`, ..., `V_n`=>`E_n`})` **raises** `[`X`]`

where E is an expression and its value is of form C`(`V'_1`=>`$N_1, ..., V'_m$`=>`N_m`)` (*i.e.*, a constructed value) and $\{V_1, ..., V_n\} \subseteq \{V'_1, ..., V'_m\}$.

The update expression returns the value of E where the fields $V_1, ..., V_n$ have been modified to values resulted from the evaluation of $E_1, ..., E_n$ expressions. If the value represented by E has not (all) the fields $V_1, ..., V_n$, the exception X is raised.

NOTE: To be sure that no exception is raised, one may wish to use field updates for record types only. In this case, the static semantics will ensure that the single constructor of the record type has the field as argument; thus, the raise clause should be omitted with current version of the compiler.

5.2.6 Unambiguous Expressions

To solve the type ambiguity introduced by the function and constructor overloading, the explicit typing of expression is allowed:

E **of** T

The evaluation of this expression is the same as the evaluation of E; the type T is used only at compile time.

The construct "(E **of** T) **raises** `[`X`]`" is used for type-subtype coercion†.

Another source of ambiguity is the precedence of Lotos NT predefined operations. This precedence can be forced using parenthesized expressions:

`(`E`)`

(E) evaluates like E. Parenthesis may be also used for esthetic considerations.

5.3 Patterns

A *pattern* is a construct allowing to obtain informations about the structure of values. The patterns, denoted by P, have the following form:

P	::=	V	*variable*	(P1)
	|	K	*constant pattern*	(P2)
	|	**any**	*wildcard*	(P3)
	|	V **as** P	*aliasing*	(P4)
	|	C [`(`PS`)`]	*constructed pattern*	(P5)
	|	`[`PS`]`	*list pattern*	(P6)
	|	P **of** T	*explicit typing*	(P7)

where PS denotes lists of pattern parameters (named for records or unnamed for tuples):

PS	::=	[**...**]	*empty or wildcard record*	(PS1)
	|	V_1`=>`$P_1, ..., V_n$`=>`P_n [**,...**]	*record*	(PS2)
	|	$P_1, ..., P_n$	*tuple*	(PS3)

The variables V belonging to a pattern P are "initialization" occurrences (*i.e.*, they should be already declared, but may be non initialized). It is not allowed to use several times the same variable V in the same pattern P.

NOTE: Like in ActOne, and unlike in functional languages, the occurrence of a variable in a pattern is a "use" occurrence and not a "define" occurrence. This design choice is compatible with the "declare before use" requirement of imperative style languages.

The pattern-matching of a value N with a pattern P has two effects:

1. Sends a boolean result which is true if N has the same structure as P, false otherwise.
2. If N matches P, the variables V used by P are initialized with the values extracted from N.

Matching is defined (recursively) as follows. Remind that patterns and values match only if they have the same type.

Pattern	Value	Condition	Effect	Result
V	N	None	V receives N	true
K	K	None	None	true
K	N	$K \neq N$	None	false
any	N	None	None	true
V **as** P	N	P and N match	V receives N	true
V **as** P	N	P and N do not match	None	false
$C(P_1, \ldots, P_n)$	$C(N_1, \ldots, N_n)$	Each P_i, N_i match	None	true
$C(P_1, \ldots, P_n)$	$C(N_1, \ldots, N_n)$	Some P_i, N_i do not match	None	false
$C(P_1, \ldots, P_n)$	N	N has not the form $C(N_1, \ldots, N_n)$	None	false
P **of** T	N	Same as matching P and N		

Note that:

- in the pattern V **as** P, V cannot occur in P;

- P **of** T is used only to solve ambiguities caused by constructor overloading.

The "**...**" notation is a shorthand meaning that all fields of the record have an "**any**" pattern. Note that the type of the record should be unambiguous. Also, the "**...**" shorthand can be used following a sequence of labelled patterns. It will be translated to a record in which the unspecified fields are "**any**" patterns.

Example 5.3.1
For a value of type `HeaderType`, the following patterns:

```
Header1 (dest, length, crc)
Header2 (dest_id => dest of int, data_length => data, header_CRC => crc, ...)
```

allow to obtain the destination (into the variable `dest`), the length of data (into the variable `length`), and the CRC (into the variable `crc`) of the value. The source value is neglected since the "**...**" are translated to `source_id => any`. ■

Note that the variables initialized by the matching of the pattern P against the value N may be used in the remainder of the description iff the pattern-matching is successful. This ensures that the variables defined inside a pattern are always initialized before use.

The patterns are mainly used in the "**case**" statement (§ 5.4.6, p. 48) and behaviour (§ 6.2.2, p. 60), but they appear also in the trap statement (§ 5.4.9, p. 53) and behaviour (§ 6.2.4, p. 62).

5.4 Statements

A LOTOS NT *statement*, denoted by I, may return a value, assign variables, and raise exceptions. The main difference between expressions and statements is that statements only may assign variables and explicitly raise exceptions.

Each statement is typed by the record of variables assigned and the value returned. The evaluation of a statement may modify the memory, return a value, or/and raise an exception.

The following grammar gives the syntax of statements:

I	::=	**return** E	*value return*	(I 1)
	|	**null**	*termination*	(I 2)
	|	V`:=`E	*assignment*	(I 3)
	|	I `;` I	*sequential composition*	(I 4)
	|	**var** $\overline{V}$`:`T[`:=`E] {`,`$\overline{V}$`:`T[`:=`E]} **in** I **end var**	*variable declaration*	(I 5)

	|	**case** E [`:`T] **is** [**var** VL **in**]
		  IM
		end case
			case statement	(I 6)

	|	**if** E **then** I
		  { **elsif** E **then** I }
		  [**else** I]
		end if
			conditional statement	(I 7)

	|	**eval** [V`:=`] F [`(`VS`)`] [**raises** `[`XS`]`]	*procedure call*	(I 8)

	| **loop** I **end loop**	*forever loop*	(I 9)
	| **loop** X **in** I **end loop**	*breakable loop*	(I 10)
	| **while** E **do** I **end while**	*while loop*	(I 11)
	| **for** I **while** E **by** I **do** I **end for**	*for loop*	(I 12)
	| **break** X	*loop break*	(I 13)
	| **raise** X $[E]$	*raise exception*	(I 14)
	| **trap** $\{$**exception** $X[:T]$ **is** $I\}$ **in** I **end trap**	*trapping exceptions*	(I 15)

$\overline{V}$ ::=	$V\ \{,V\}$	*list of variable identifiers*	(VL1)
VL ::=	$\vec{V}:T\ \{,\vec{V}:T\}$	*variable list*	(VL2)

IM ::=	P {`|` P} [`[`E`]`] `->` I	*match-statement*	(IM1)
|	IM `|` IM	*list*	(IM2)

VE ::=	E	*actual parameter "in"*	(VE1)
|	`?`V	*actual parameter "out" and "inout"*	(VE2)

VS ::=	[`...`]	*empty or wildcard*	(VS1)
|	$V \Rightarrow VE\ \{,V \Rightarrow VE\}$ [,`...`]	*disjoint union*	(VS2)
|	$VE\ \{,VE\}$	*list*	(VS3)

where IM are match instructions, VE are actual value parameter, and VS are sequences of actual value parameters.

In the following we present each Lotos NT statement.

5.4.1 Value return

The evaluation of the statement "**return** E" begins with the evaluation of E. If E evaluates successfully, the statement returns the value of E. If E raises an exception, the statement raises the same exception and terminates unsuccessfully (blocks).

5.4.2 Null Statement

The statement "**null**" has no other effect than termination. It does not return any value and it does not assign any variable.

5.4.3 Assignment

The effect of an assignment statement "V`:=`E" is the modification of the value stored by the variable V at the value given by the expression E. Note that lateral side effects are avoided because the expression E cannot assign other variables. It can only return a value.

The evaluation starts by evaluating E. If E terminates successfully, the resulting value is assigned to the variable V. If E raises an exception (§ 5.4.9, p. 52), the exception is propagated and V is not assigned.

The value of a variable may also be modified by function call (§ 5.5.2, p. 56) or process call (§ 6.7, p. 66). A variable can take several successive values, for example:

```
V := 0 ; V := V + 1
```

where the variable `V` receives values 0 and 1. As long as statements and expressions cannot have a behaviour, the variable `V` takes these values at the same instant.

5.4.4 Sequential Composition

In the sequential composition of statements "I_1 `;` I_2", the statement I_1 cannot return a value but may assign variables (*i.e.*, the return statement should be in the final position of the sequential composition).

The evaluation starts by evaluating I_1. If I_1 terminates successfully, the result is given by the evaluation of I_2. If I_1 raises an exception (§ 5.4.9, p. 52), the exception is propagated and I_2 is never started. For example "`raise X; V:=1`" will never assign 1 to `V`.

5.4.5 Variable declaration

Variables may be declared (and initialized) using the local variable declaration statement, which has the simple form:

var V_1`:`T_1[`:=`E_1]$, ..., V_n$`:`T_n[`:=`E_n]
in I
end var

where $V_1, ..., V_n$ are variable identifiers, $T_1, ..., T_n$ are type identifiers, $E_1 ..., E_n$ are expressions, and I is a statement.

A "**var**" statement declares the names of variables having the same scope, their types, and (optionally) their initial values. The scope of a variable declaration is the body I. Scoping is lexical: any redeclaration of a variable hides the outer declaration.

Variables $V_1, ..., V_n$ should be different. The types of expressions $E_1 ..., E_n$ should be *resp.* $T_1, ..., T_n$.

5.4.6 Case statement

Lotos NT allows to describe conditional processing of data by using constructs similar to those used by the usual programming languages: "**case**" and "**if**".

The most general conditional statement offered by Lotos NT is the "**case**" statement, whose simplest form is:

```
    case E_0 : T is
        var V_1 : T_1 , ... , V_n : T_n in
        P_1 [E_1] -> I_1
      | ...
      | P_n [E_n] -> I_n
    end case
```

where $n \geq 1$. The expression E_0 must have the same type as the patterns $P_1, ..., P_n$. The expressions $E_1, ..., E_n$ must be boolean *guards*, and their default value is **true**. The statements $I_1, ..., I_n$ should return a value of the same type and initialize the same set of (non-local declared) variables. This condition is important for the control of the variable flow. The scope of variables $V_1, ..., V_n$ are the patterns P_i, the boolean guards E_i, and the statements I_i.

The patterns $P_1, ..., P_n$ must be exhaustive, *i.e.*, they must cover all the possible values of type T. There exists algorithms that check statically this condition [Sch88]. To make a list of patterns exhaustive one can add a clause "**any of** T `->` I_{n+1}" at the end of the list.

The evaluation of a "**case**" statement is made as follows. Let N_0 be the value of the expression E_0; N_0 is matched sequentially over the clauses corresponding to patterns $P_1, ..., P_n$ until it matches one. The value N_0 matches a clause "P_i `[`E_i`]`" if N_0 matches P_i and the evaluation of E_i in the context of variables bound by the matching returns **true**. The result of the case statement is the same as the result of the statement I_i (evaluated in the context of variables bound by P_i) corresponding to the first clause i which matches N_0.

Example 5.4.1
The statement below returns the destination identifier of an expression E of type `HeaderType`:

```
case E of HeaderType is
  var dest: int in
    Header1 (dest, any, any) -> return dest
|   Header2 (dest => dest, ...) -> return dest
end case
```

Note that the patterns cover all the values of type `HeaderType`. The wildcard "`any`" are used to match all values which are not interesting for the remainder of the statement. ■

Note that constant values may be filtered by giving their values, excepting the floating point values (of `real` type). Constants may also be filtered by using the "**if**" statement.

A more sophisticated form of the "**case**" statement provides factorization of clauses which have the same target statement I_i. Consider for example, the statement which encodes the working days of a week by 1 and the week-end days by 0 using the above "**case**" statement:

```
case E: day_of_week is
    Monday    -> return 1
|   Tuesday   -> return 1
|   Wednesday -> return 1
|   Thursday  -> return 1
|   Friday    -> return 1
|   Saturday  -> return 0
|   Sunday    -> return 0
end case
```

A simpler form with the same effect is:

```
case E is
    Monday | Tuesday | Wednesday | Thursday | Friday -> return 1
|   Saturday | Sunday -> return 0
end case
```

A value N_0 matches a clause "P_i^0 `|` ... `|` $P_i^{m_i}$`[`[E_i]`]`" iff one of the pattern P_i^j matches, and the guard E_i evaluated in the context of variables bound by P_i^j returns **true**.

5.4.7 If statement

The "**if**" construct allows conditional computations; it is generally included in all languages. In Lotos NT it has the form:

if E_0 **then** I_0
 elsif E_1 **then** I_1
 ...
 elsif E_n **then** I_n
 [**else** I_{n+1}]
end if

where $n \geq 0$, so the "**elsif**" clauses are optional. The default "**else**" clause is "**exit null**".

The expressions $E_0, ..., E_n$ are called *conditions*. They must be of type `bool`, and do not have side effects. The statements $I_0, ..., I_{n+1}$ should return a value of the same type and should initialize the same variables. This constraint is important for the control of the variable flow.

The evaluation of an "**if**" statement is done as follows. The conditions $E_0, ..., E_n$ are evaluated in this order until a condition E_i evaluates to **true**; the evaluation of the "**if**" statement is the same as the evaluation of I_i. If all conditions are false, the result of "**if**" is the result of I_{n+1}, which by default is "**null**".

Example 5.4.2
The following statement computes the maximum of two integral numbers `X` and `Y`:

```
if X >= Y then return X else return Y end if
```

■

The "**if**" statement is less powerful than the "**case**" statement. Moreover, the "**if**" statement above may be translated into the following (equivalent) "**case**" statement:

```
case $E_0$ is
   true -> $I_0$
 | false ->
       case$E_1$ is
          true -> $I_1$
        | false -> ...
       end case
end case
```

However, in this case it is recommended to use the "**if**"" statement instead of the sophisticated "**case**" statement.

5.4.8 Iteration Statements

LOTOS NT allows to describe repetitive processing of data by mean of several iteration constructs. The most general and simplest one is the unbreakable loop, but several specialized iteration constructs are also provided, like breakable, conditional ("**while**"), and iterative ("**for**") loops.

Iterative constructs provide a way to express recursive processing of data without use of recursive functions. This avoids the stack overflow due to the infinite or great number of recursive function calls.

Loop forever statement The simplest iterative construct offered by LOTOS NT is the "**loop**" forever statement:

```
loop $I$ end loop
```

where I is called the loop *body*.

The evaluation of a "**loop**" forever statement never terminates. The statement I is repeatedly evaluated. It can read and write variables of the current context. This type of iteration may introduce non-terminating processing of data. This may be signaled by the compiler. Note that if I raises a handled exception (§ 5.4.9, p. 52), the loop is interrupted.

Breakable loop statement In practice, it is difficult to imagine examples of data processing which never terminate. Statements are generally used to compute (instantaneously) values. For this reason a form of breakable loop is provided:

```
loop $X$ in
  $I$
end loop
```

where X is the loop identifier, *i.e.*, the loop name. The statement I may read and write variables of the current context with respect to static semantics constraints.

A loop is broken using the "**break**" statement which has the following syntax:

```
break $X$
```

where X is the name of the loop to be broken. Note that "**break**" cannot interrupt other types of loops ("**while**" and "**for**").

Example 5.4.3
The following statement computes the sum of elements of a given list of integers `xs`:

```
var ys: intlist := xs, total: int := 0
in
    loop Sum in
        case ys is
            var z: int, zs: intlist in
            nil -> break Sum
        |   cons (z, zs) -> total := total + z;
                            ys := zs
        end case
    end loop
end var
```

■

The named loops are used to break loops which are not the inner one, for example:

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

As will be shown in Section 5.4.9, the breakable loop is a syntactic sugar for infinite loop construct and exception handling.

While statement The conditional execution of a loop may be expressed using the "**while**" construct which exists in most languages:

while E_0 **do**
  I
end while

where E_0 is an expression of type `bool`. I is the body of the loop, which may return a result (not used), read or write the variables of the current context.

At each iteration the expression E_0 is evaluated; if it returns **true**, the statement I is evaluated; otherwise, the "**while**" statement terminates. This loop cannot be interrupted by "**break**" since no name is specified.

Example 5.4.4
The statement below computes the factorial of `n`:

```
var k: int := n,
    fact: int := 1
in
    while (k > 0) do
        fact := fact * k;
        k := k - 1
    end while;
    return fact
end var
```

The property `fact` = `n`!/`k`! is the invariant of the loop, and the termination is ensured by the fact that `k` decreases at each iteration. The result of the statement is `fact` = `n`!/0! = `n`!. ■

This form of loop may be translated into a breakable loop as follows:

```
loop $X$ in
  if $E_0$ then $I$
  else break $X$
  end if
end loop
```

For statement The last iterative construct, the "**for**" statement, allows to describe in a compact form finite iterations. Its form is closed to the "**for**" construct of the C language:

```
for $I_0$ while $E_1$ by $I_2$ do
  $I$
end for
```

where I_0 is a statement doing only variable assignments; E_1 is an expression of type `bool`; I_2 is a statement doing only assignments; I is the body of the loop, repeatedly executed. It can assign variable but it cannot return a result.

The evaluation of a "**for**" statement begins with evaluating the initialization statement I_0 in the current context of variables. Then, while the boolean expression E_1 evaluates to **true**, the body of the loop, I, is evaluated and when I terminates then I_2 is evaluated. If E_1 evaluates to **false**, the "**for**" statement terminates. This loop cannot be interrupted by "**break**" since no name is specified.

In fact, this form of loop is syntactic sugar of breakable loop; it can be translated as follows:

```
$I_0$;
loop $X$ in
  if $E_1$ then $I$; $I_2$
  else break $X$
  end if
end loop
```

5.4.9 Exceptions and their handling

An important feature of the language is the possibility to signal and treat the errors or unexpected cases by raising and trapping exceptions.

Raise statement Exceptions are used to interrupt the evaluation of expressions or statements. An exception may be raised using the "**raise**" statement, whose simplest form is:

raise X

X is an exception identifier which should be already declared. The declaration of an exception is made using the "**trap**" statement.

The evaluation of a "**raise**" signals the exception X and blocks the evaluation.

Example 5.4.5
The following specification raises the exception `Hd` if one tries to take the head of an empty list:

```
case xs: intlist is
    var x: int in
    nil -> raise Hd
|   cons (x, any) -> return x
```

```
end case
```

■

Exceptions may carry values. The syntax of a valuated exception raising is:

raise X E

This form of exception raising is useful to pass a value to the handler (§ 5.4.9, p. 53).

Example 5.4.6
The presence of an unknown error code may be signaled by sending the value of the code:

```
if (error_code == 0) then
    return "minor error"
elsif (error_code == 1) then
    return "major error"
else
    raise Unknown error_code
end if
```

■

Trap statement Exceptions either propagate to top level, or are *trapped* by a "**trap**" construct containing the exception handler. The simplest syntax of the "**trap**" construct is:

```
trap
  exception $X_1$ [$V_1$:$T_1$] is $I_1$
  ...
  exception $X_n$ [$V_n$:$T_n$] is $I_n$
in
  $I_0$
end trap
```

where $n \geq 0$. The clauses contained between keywords "**trap**" and "**in**" are called *exception handlers*. An exception handler declares the name of the exception X_i, its type T_i (by default `none`), and its treatment I_i.

The scope of exception identifiers $X_1, ..., X_n$ is only the statement I_0. So exceptions $X_1, ..., X_n$ are handled only if raised by I_0. If one of $I_1, ..., I_n$ raises an exception X_i, it is not handled by the current "**trap**" statement.

The statements $I_1, ..., I_n$ may either return a value of the same type and initialize the same variables as the statement I_0, or block. These constraints are checked statically. They ensure that the "**trap**" statement is well typed, and the flow of initialized variables is the same whatever the evolution of evaluating I_0 is.

The evaluation of the "**trap**" statement begins with the evaluation of I_0. If I_0 raises one of the exceptions X_i, its evaluation is interrupted, and the result of the "**trap**" statement is the result of I_i. If I_0 terminates normally, *i.e.*, without raising any of the exceptions X_i, the "**trap**" statement also terminates.

Example 5.4.7
The following statement returns 0 when `c` equals 0, or assigns the value of `b/c` to `a` otherwise.

```
trap
    exception ZD is return 0
in
```

```
    a := (b/c) raises [ZD]
end trap
```

■

Example 5.4.8
The breakable loop below:

```
loop X in
  I
end loop
```

is a syntactic sugar for an infinite loop broken by an exception raising:

```
trap
  exception X is null
in
  loop I end loop
end trap
```

The statement "**break** X" is syntactic sugar for "**raise** X". ■

The "**trap**" statement both declares and traps the exception—this means it is impossible for an exception to escape outside its scope.

NOTE: This can be contrasted with a language such as SML where exception declaration and handling are separated, so it is possible for exceptions to escape their scope.

```
local
  exception Foo
in
  raise Foo
end
```

Note that the only way in which an exception can be observed by its environment is by trapping it.

Implementation issues If the code generated by Traian is used in external implementations, the C function `TRAIAN_INIT ()` must be called before any action. This function initializes the structures used by Traian for the implementation of the exception mechanism.

5.5 Functions

Functions are a mean for code structuring and re-usability.

This section describes how Lotos NT users may define and use functions. The Lotos NT predefined functions are described in Section (§ 4.3, p. 29).

5.5.1 Function definition

A function definition has the following syntax:

```
function F ([A_1] V_1:T_1, ..., [A_n] V_n:T_n) [:T]
      [ raises [X_1:T_1, ..., X_m:T_m] ]
  is I
end function
```

$$\textbf{function}\ F\ (\texttt{[}A_1\texttt{]}\ \overline{V}_1 : T_1,\ \ldots,\ \texttt{[}A_n\texttt{]}\ \overline{V}_n : T_n)\ [:T]\ [\ \textbf{raises}\ \texttt{[}\overline{X}_1 : T_1,\ \ldots,\ \overline{X}_m : T_m\texttt{]}\ ]\ \textbf{is}\ I\ \textbf{end function}$$

where A_i may be "**in**", "**out**", or "**inout**". The default value for A_i is "**in**".

NOTE: This form for function declaration was chosen for syntactic compatibility with IDL (and Ada) and for an easier interface with C.

F is the name of the function. Two function names may be the same if their profiles (*i.e.*, the types of parameters or the result type) differ.

($[A_i]$ $\overline{V}_1$:T_1, ...) is the list of *formal value parameters*. Value parameters may be constant values ("**in**" parameter), result values ("**out**" parameter), or modifiable values ("**inout**" parameters); the default type is "**in**". An "**in**" parameter may be read but its value is not changed by the function call, although its value may be changed by I. An "**out**" parameter should be assigned by I and its value is visible after the function call. An "**inout**" parameter has an initial value, and I may modify them; the value of the parameter assigned by I is visible after the function call. The scope of variables in the lists $\overline{V}_1, ..., \overline{V}_n$ is the body of the function, I.

T is the result type of the function.

[$\overline{X}_1$:T_1, ...] is the list of *formal exception parameters*. The scope of the exceptions in the lists $\overline{X}_1, ..., \overline{X}_m$ is the body of the function, I.

The statement I computes the result value of the function and the output parameters. Its environment is the list of formal parameters (value and exception). In Lotos NT it is not possible to assign "global" variables or to raise "global" exceptions. All variables and exceptions used by the body of the function must be declared as function parameters. If T is given, the result type of I must be T. The values assigned to output parameters must be correctly typed.

Example 5.5.1
Consider the declaration of the function `hd`:

```
function hd (xs: intlist) : int raises [Hd: none]
is
    case xs is var x: int in
        nil -> raise Hd
    |   cons (x, any) -> return x
    end case
endfun
```

where `xs` is an input parameter of type `intlist`. Note that the body of the function does not assign global variables, the variable `x` being local to the second clause of the "**case**" statement.

The declaration below uses the output parameters to return several results:

```
function partition (in xs: intlist, out less: intlist, in x: int, out gtr: intlist)
is
    var ys: intlist := xs,
        ls: intlist := nil,
        gs: intlist := nil
    in
        loop P in
            case ys is var z: int, zs: intlist in
                nil -> break P
            |   cons (z, zs) ->
                    if (z < x) then
                        ls := cons (z, ls)
                    else
                        gs := cons (z, gs)
                    end if;
                    ys := zs
               end case
          end loop;
          less := ls;
          gtr  := gs
     end var
end function
```

■

5.5.2 Function call

Function calls can be used in both expressions and statements. In expressions, functions that have two parameters can be used either in the prefix form or in the infix form.

The call of a function F in Lotos NT has two forms. The "positional" function call give the ordered list of the parameters:

eval $[V$ `:=`$]$ F `(`$[$`?`$V_1'|E_1]$`, ..., `$[$`?`$V_n'|E_n]$`)` **raises** `[`X_1'`, ..., `X_m'`]`

where n, *resp.* m, must be equal to the number of formal value parameters, *resp.* to the number of formal exception parameters. `(`$[$`?`$V_1'|E_1]$`,...)` is the list of *actual value parameters*. Expressions E_i should appear in the same position as the "in" parameters and must have the same type. Variables V_i should appear as actual parameters of the "inout" and "out" formal parameters, and must be already declared with the same type as the formal parameters. $X_1', ..., X_n'$ are *actual exception parameters*. The result of the function, if any, may be assigned to the variable V.

The "named" call of the functions use the name of formal parameters to specify the correspondence between formal and actual parameters; the order of the actual parameters is not important. The three alternatives below are equivalent:

eval $[V$ `:=`$]$ F `(`V_1`=>`$[$`?`$V_1'|E_1]$`, ..., `V_n`=>`$[$`?`$V_n'|E_n]$`)` **raises** `[`X_1'`, ..., `X_m'`]`

where the list of actual value parameters is named,

eval $[V$ `:=`$]$ F `(`$[$`?`$V_1'|E_1]$`, ..., `$[$`?`$V_n'|E_n]$`)` **raises** `[`X_1`=>`X_1'`, ..., `X_m`=>`X_m'`]`

where the list of actual exception parameters is named, and

eval $[V$ `:=`$]$ F `(`V_1`=>`$[$`?`$V_1'|E_1]$`, ..., `V_n`=>`$[$`?`$V_n'|E_n]$`)` **raises** `[`X_1`=>`X_1'`, ..., `X_m`=>`X_m'`]`

where both lists are named. The constraints above are also applied here.

Note that the "positional" style cannot be merged with the "named" style in the same list of actual parameters. This style of function call is similar to the Ada style.

The "**...**" shorthand for the record of actual parameters VS is expanded to the list of unspecified parameters as follows: if the parameter is an "in" one, the expression is the variable which has the same name as the formal parameter; if the parameter is an output ("**inout**" and "**out**"), the actual parameter is the query symbol followed by the name of the formal parameter. Similarly for the list of actual exception parameters.

The evaluation of a function call begins with the left-to-rigth evaluation of expressions corresponding to the input parameters. For the "inout" parameters, the input value is the value of the variable given as parameter. Then, the body of the function is evaluated in the context of actual values for input parameters and of actual exception parameters. The body should assign all the "out" parameters and should return a value if the function returns a value.

Note that LOTOS NT supports call by value ("in" parameters) and (a restricted form of) call by reference ("inout" and "out" parameters). Functions as arguments of functions (second order functions) are not allowed.

Example 5.5.2
The function `partition` may be used by a quick sort function as follows:

```
function quicksort (xs: intlist) : intlist
is
    case xs is var y: int, ys: intlist in
        nil -> nil
    |   cons (y, ys) ->
            var l: intlist, g: intlist
            in
                eval partition (xs, ?l, y, ?g);
                return append (quicksort (l), cons (y, quicksort (g)))
            end var
     end case
end function
```

Note that the variables `l` and `g` are locally declared to the second clause of the case because the first clause does not initialize them. ■

Chapter 6

Behaviour Expressions and Processes

In a language dedicated to the description of behaviours of (information processing) systems, the *behaviour expressions* are the most important concept. This chapter describes how LOTOS NT supports the description of behaviours and the definition of processes. It introduces important notions as: concurrency, communication, non-determinism, signaling, exceptions and exception handling, real-time.

Although behaviour expressions use the expressions of the data part of the language, this chapter does not make any assumption about the form of these expressions. It supposes only that expressions of the data part may be evaluated into *values*, *i.e.*, the terms of the domains of the types defined into the specification, or may assign variables. This assumption allows to interface the control part of LOTOS NT with any language expressing data expressions. Chapter 5 presents the data language supported by LOTOS NT. The behaviour language can be seen as an extension of the data language with concurrency, non-determinism, and real-time features.

Some constructs of the language use bracketed keywords: "**par—end par**", etc. The unbracketed constructs, like "`[]`" are left associative. To resolve remaining syntactic ambiguities, any behaviour can be bracketed using parenthesis "`()`". LOTOS NT is fully orthogonal: operators can be freely mixed in an arbitrary way. One can compose parallel behaviours sequentially, or put sequences in parallel, one can subject any behaviour to a trapping, etc.

6.1 Basic Behaviours

There are four basic pure behaviours:

block
stop
null
wait (E)

The "**block**" behaviour is an inactive behaviour, which cannot do any communication, never terminates, and stops time evolving. The "**stop**" behaviour pauses forever and never terminates nor does any communication. The "**null**" behaviour terminates instantaneously when started. The "**wait**" behaviour terminates after the elapsing of the time given by the evaluation of the expression E. E

must be of type `time`.

Note that the "**block**" behaviour is the one of ET-LOTOS, the "**stop**" behaviour already exists in LOTOS but in its untimed version, and the "**null**" behaviour corresponds to the unvalued "**exit**" behaviour of LOTOS.

6.2 Regular Behaviours

As already mentioned, the control part and the data part of LOTOS NT are symmetrical: behaviours are extensions of statements. The constructs which are similar in both languages are grouped in the generic class of regular behaviour expressions.

The regular behaviour expressions are: variable declaration ("**var**"), conditional behaviours ("**case**" and "**if**"), loop behaviours (infinite and breakable "**loop**", "**while**", and "**for**"), exception raising and handling ("**raise**" and "**trap**").

In order to avoid repetition, the regular behaviour expressions will be presented shortly. For more details, the reader must refer to the corresponding section of Chapter 5.

6.2.1 Variable declaration

Variables may be declared (and possibly initialized) using the local variable declaration behaviour, which has the form:

```
var V1:T1[:=E1], ..., Vn:Tn[:=En]
in B
end var
```

where $V_1, ..., V_n$ are variable identifiers, $T_1, ..., T_n$ are type identifiers, and $E_1 ..., E_n$ are expressions. The scope of a variable declaration is the body B.

See Section (§ 5.4.5, p. 47) for more details.

6.2.2 Conditional behaviour

The most general conditional expressions offered by LOTOS NT is the "**case**" expression, whose simplest form is:

```
case E0:T is
    var V1:T1, ..., Vn:Tn in
    P1 [E1] -> B1
  | ...
  | Pn [En] -> Bn
end case
```

where $n \geq 1$, $P_1, ..., P_n$ are patterns (see (§ 5.3, p. 44)), and $E_1, ..., E_n$ are boolean guards (see (§ 5.4.6, p. 48)). The behaviours $B_1, ..., B_n$ (which substitute statements $I_1, ..., I_n$ in 5.4.6) must initialize the same set of variables if they terminate. The scope of variables bound by a pattern P_i are the boolean guards E_i and the behaviours B_i.

The "**case**" behaviour has the same semantics and needs the same constraints as the corresponding "**case**" statement (§ 5.4.6, p. 48).

The "**if**" construct allows conditional computations; it is also called *deterministic* selection. It has the following form:

```
if $E_0$ then $B_0$
 elsif $E_1$ then $B_1$
 ...
 elsif $E_n$ then $B_n$
 [else $B_{n+1}$]
end if
```

where $n \geq 0$, *i.e.*, the "**elsif**" clauses are optional. The "**else**" clause has the default value "**exit null**".

Behaviours $B_0, ..., B_{n+1}$ play the same role as expressions $I_0, ..., I_{n+1}$ in (§ 5.4.7, p. 49). They must initialize the same variables if terminate. This constraint is important for the control of the variable flow.

The "**if**" behaviour has the same semantics and needs the same constraints as the corresponding "**if**" statement (§ 5.4.7, p. 49).

6.2.3 Iteration behaviour

The iteration constructs provided by the behaviour language are similar to those of the data language (§ 5.4.8, p. 50). The body of the loop is a behaviour and not a statement.

The loop forever behaviour has the form:

```
loop $B$ end loop
```

where B is called the loop *body*. B is repeatedly executed after its termination.

The breakable loop behaviour has the form:

```
loop $X$ in
  $B$
end loop
```

This loop is broken by the "**break**" behaviour:

```
break $X$
```

which breaks the loop X and blocks.

The conditional execution of a loop may be expressed using the "**while**" construct:

```
while $E_0$ do
  $B$
end while
```

B is executed while the boolean condition E_0 returns **true**.

The "**for**" iteration has the syntax:

```
for $I_0$ while $E_1$ by $I_2$ do
  $B$
end for
```

where B is executed repeatedly after the initializations I_0, while E returns **true**. At the end of each iteration, the statement I_2 is executed (it should contain only assignments).

6.2.4 Exception raising and handling

In the control part, the exceptions are a special case of signals (see Section 6.6). The behaviour raising an exception:

raise X $[E]$

is equivalent to signaling X with the value given by evaluating E, followed by a "**block**" behaviour ("**signal** X $[E]$; **block**"). So, raising an exception blocks (time cannot pass) the current behaviour after the exception signaling.

An exception (or a signal) either propagates to top level, or is *trapped* by the "**trap**" behaviour containing the exception handler. The syntax of the "**trap**" behaviour is:

trap
  exception X_1 :T_1 **is** B_1
  ...
  exception X_n :T_n **is** B_n
in
  B_0
end trap

where $n \geq 0$. The constraints given in Section 5.4.9 must be respected, where the statements $I_0, I_1, ..., I_n$ are substituted with the behaviours $B_0, ..., B_n$ respectively.

6.3 Assignment and Procedure Call

The assignment behaviour extends the assignment statement defined in Section (§ 5.4.3, p. 47). Assignment behaviour may be deterministic or non-deterministic. They terminate *instantaneously*.

The deterministic assignment behaviour has the form:

V := E

where V is a variable (§ 5.4.5, p. 47), and E is a data expression. The variable and the expression must have the same type. The value stored by the variable V is given by the expression E.

Example 6.3.1
The behaviour below declares the variable `V` of type **time** (whose domain is discrete) and waits 15 time units:

```
var V : time
in
    for V:=0 while V <= 5 by  V:=V+1 do
       wait (V)
    end for
end var
```

■

The non-deterministic assignment behaviours have the form:

V := **any** T [[E]]

where V is a variable, T is a type (the same as the declared type for V), and E is a data expression of type `bool`, also called *boolean guard*. The default value for the guard is "[**true**]". The non-deterministic assignment assigns to V a value of the domain of T which satisfies the guard E. It corresponds to the logical formula:

$\exists V : T. \quad E \Longrightarrow ...$

Example 6.3.2
The behaviour below declares the variable `V` of type `time` (whose domain is discrete) and waits a while, but strictly less than five time units, i.e. no more than four.

```
var V : time
in
    V := any T [V < 5];
    wait (V)
end var
```

■

The behaviour function call has the same form as the function call statement (§ 5.5.2, p. 56). For example:

eval V `:=`F `(`[`?`V_1'|E_1]`,` ...`,` [`?`V_n'|E_n]`)` **raises** `[`X_1'`,` ...`,` X_m'`]`

is a positional call of function F. The same constraints as mentioned in 5.5.2 for the actual parameters are available. The function call is instantaneous.

6.4 Communication and Actions

In Lotos NT, as in Lotos, the behaviours communicate by *rendezvous* on *gates*. If G is a gate, the behaviour:

G

is the simplest one which communicates on G. It waits for the rendezvous on gate G, and than terminates.

The time at which the communication takes place can be obtained using the annotation "`@`", as follows:

G `@`V [`[`E`]`]

where V is a variable of type `time`, and E is an expression of type `bool`. E is also called *boolean guard*. The variable V can appear in the boolean guard E.

The behaviour waits for a communication on gate G till the guard E evaluates to **true**. At any moment, V gives the time elapsed from communication enabling. After communication, the behaviour terminates immediately. If the communication cannot take place, the behaviour has the same semantics as "**stop**".

Example 6.4.1
The following behaviour waits for a rendezvous on gate G at most 3 time units:

```
G @t [t < 3]
```

where `t` is a variable of type `time`.

The following behaviour may execute a rendezvous on G at 3 time units after its beginning:

```
G @t [t == 3]
```

■

This simple form of rendezvous does not allow to send or receive data, so it is a *pure* communication. The most general form of communication allows for data exchange over the gates with the following syntax:

$G[.C]$ `(`O_1`, ...,` O_n`)` $[$`@`$V]$ $[$`[`E`]`$]$
$G[.C]$ `(`V_1`=>`O_1`, ...,` V_n`=>`O_n`)` $[$`@`$V]$ $[$`[`E`]`$]$

where $n \geq 0$, and O is an *offer*. Offers have two forms:

O	::=	`!`E	*send value offer*	(O1)
	|	`?`P	*receive value offer*	(O2)

The offer "`!`E" corresponds to an emission over the gate G of the value of expression E. The offer "`?`P" corresponds to the reception over the gate G of a value which is matched against P. The variables of P must be already declared.

NOTE: It is possible to introduce the same syntactic sugar as in LOTOS, *i.e.*, "`?`V`:`T" meaning the declaration and the initialization of V. This form is closer to patterns, but it goes against the "imperative" style of the language.

The gates in LOTOS NT must be typed. So the offers are also typed, and their types must correspond to the type of the gate. NOTE: The use of types for typing gates does not allow the same syntactic means to express communication as channels (multiple profiles and in/out) or sub-typing (multiple profiles).

The communication is blocked by both sending and receiving values: the behaviour waiting for a rendezvous is suspended (time can pass) and terminates immediately after the rendezvous takes place. The rendezvous takes place if the values exchanged match and the boolean guard "`[`E`]`" evaluates to **true**.

In LOTOS NT, the rendezvous is symmetrical: there is no difference between the sender and the receiver. The rendezvous on a gate may allow several sending and offers at the same time. Moreover, the same gate may be used in several rendezvous either with a receive offer, or with a send offer.

Example 6.4.2
The following behaviour accepts rendezvous on `LINK_DATA_recv`, with 0 as destination field, after 5 time units:

```
LINK_DATA_recv (dest_id => !0, source_id => ?src, data => ?d) @t [t > 5]
```

The variables `src` and `d` are respectively assigned the values of fields `source_id` and `data`. The gate `LINK_DATA_recv` has the type `LinkPacketType` defined below:

```
type LinkPacketType is
    record dest_id: int, source_id: int, data: LinkDataType
end type
```

The header of a packet is communicated by the Link layer to a Physical layer over the gate `Physical_Data` which has the type `HeaderType`. The behaviour below accepts rendezvous on `Physical_Data` where the header contains a source field of value 3:

```
Physical_Data.Header2 (source_id => !3, dest_id => any,
                       data_length => any, header_CRC => any)
```

■

Gates are declared using the "**hide**" operator (§ 6.10, p. 74), the "**rename**" operator (§ 6.11, p. 76), by a process declaration (§ 6.7, p. 66), and by the local declarations of a specification (§ 7.5, p. 89).

There are two special gates in LOTOS NT: "**i**" and "δ". "**i**" is a keyword of the language, also called the *internal* gate. It specifies an internal action of the behaviour, and it terminates immediately. It cannot have offers or guard. "δ" is also called *termination* gate. It does not appear explicitly in the language, but it appears each time a behaviour terminates. For example, the "**null**" behaviour does only a rendezvous on the δ gate. The termination gate plays an important role on synchronization (§ 6.9, p. 70).

6.5 Sequential Behaviours

Like for statements 5.4.4, the sequencing of behaviours is done by "`;`" sequential composition operator:

B_1 ; B_2

where B_1 and B_2 are behaviours.

The first behaviour B_1 is instantaneously started when the sequence is started. If B_1 terminates, B_2 is immediately started and the sequence behaves as B_2 from then on. If B_1 stops, blocks, or raises an exception, B_2 is never started.

Example 6.5.1
The following behaviour waits for a rendezvous on gate `MONEY` (receives the money) and then a rendezvous on gate `TEA` (distributes a cup of tea), and terminates:

```
MONEY; TEA
```

A one place buffer of integers which communicates on gates `input` and `output` has the following behaviour:

```
var x: int
in
    input (?x); output (!x)
end var
```

The gates `input` and `output` should have type `int`. ■

The LOTOS NT operator "`;`" is associative and accepts "**null**" as neutral element.

Note that the sequential composition may be seen as a synchronization on "δ": as soon as the behaviour B_1 does a rendezvous on δ (*i.e.*, terminates), the behaviour B_2 is started. The termination of B_1 and the beginning of B_2 are atomic; δ does not appear as an action of the sequential behaviour. This is similar to the action prefix operator "`;`" of LOTOS, but differs from the LOTOS sequencing operator "`>>`" which introduces an internal action.

6.6 Signaling

LOTOS NT introduces a new class of observable actions, called *signals*. Signals are an extension of exceptions used in the data part (§ 5.4.9, p. 52). They are used to broadcast an *urgent* information throughout the behaviour which is the scope of the signal. A signal may carry value information, so it is a typed object. The signals of LOTOS NT are similar to the "signal" model of Timed CSP [Sch93].

Instantaneous signal emission is realized by the "**signal**" behaviour whose general form is:

signal X $[E]$

where X is a signal identifier. The expression E must have the same type as the declared type of X. The expression E is not present if X is of type `none`.

The "**signal**" behaviour evaluates the data expression E to N (if exists), does action X N, and terminates instantaneously. Since the evaluation of E does not take any time, the "**signal**" behaviour has no time evolving.

Signaling is similar to raising exception ((§ 5.4.9, p. 52) and (§ 6.2.4, p. 62)) except that it allows computation to carry on after the raising of the signal: "**raise** X" is a shorthand for "**signal** X **;** **block**.

Apart from exception raising, signals are also a mean to broadcast urgent information about a specific internal action occurrence (see Section 6.10 for more details).

Signals, like exceptions, may be declared by the "**trap**" behaviour (§ 6.2.4, p. 62), by the "**rename**" behaviour (§ 6.11, p. 76), by the process declaration (§ 6.7, p. 66), and by the local declarations of a specification (§ 7.5, p. 89).

6.7 Processes Definition and Instantiation

LOTOS NT (like LOTOS) allows to name a behaviour using a *process* definition. A process is an object which denotes a behaviour; it can be parameterized by a list of formal gates, a list of formal variables, and a list of formal signals (exceptions). Note that processes, like functions, cannot be parameters of processes: LOTOS NT is a first order language.

6.7.1 Process definition

A process definition has the following syntax:

> **process** Π [[G_1:T_1, ..., G_p:T_p]]([A_1] V_1:T_1, ..., [A_n] V_n:T_n)
>   [**raises** [X_1:T_1, ..., X_m:T_m]]
>   **is** B
> **end process**

It declares the process Π (possibly parameterized by the formal gates $G_1, ..., G_p$, the formal variables $V_1, ..., V_n$, and the formal signals $X_1, ..., X_m$) whose body is the behaviour B. The default formal variables are "**in**".

NOTE: In LOTOS, gates are not typed. One may see this like typing gates by a type "`any`" which is a super-type of all types. In LOTOS NT, gates are strongly typed.

Process names cannot be overloaded. The name Π must be unique.

The formal variables must respect the same constraint as for functions (§ 5.5.2, p. 56).

The behaviour B may be any behaviour which initializes all the "**out**" parameters. It is executed in the context built from the formal parameters, the declared types, functions, and processes. In LOTOS NT it is not possible to assign "global" variables or to raise "global" exceptions. All gates, variables, and exceptions used in the body of the function must be declared as parameters or declared as local in the body.

Example 6.7.1
The process below declares a generic task uniquely identified by the `id` parameter, and which executes an activity taking `d` time units. Before beginning it does a rendezvous on the gate `start`. It indicates the end of its activity by doing a rendezvous on the gate `end`.

```
process Task [start, end: int] (id: int, d: time)
is
    start !id; wait (d); end !id; stop
end process
```

■

6.7.2 Process instantiation

The instantiation of a process is done by substituting the actual parameters to the formal ones. The simplest form is the "positional" (§ 5.5.2, p. 56) process instantiation which gives the ordered list of actual parameters:

$$\Pi \ [\texttt{[}G'_1, ..., G'_p\texttt{]}] \ [\texttt{(}[\texttt{?}V'_1|E_1]\texttt{,} \ ...\texttt{,} \ [\texttt{?}V'_n|E_n]\texttt{)}] \ [\ \textbf{raises} \ \texttt{[}X'_1, ..., X'_m\texttt{]}]$$

where p, n, and m must be respectively equal to the number of formal gate parameters, the number of formal value parameters, and the number of formal exception parameters. The actual parameters must have the same type as the formal ones. Expressions E_i should appear at the same position as the "**in**" parameters and must have the same type. Variables V_i should appear as actual parameters of the inout and out formal parameter, and must be already declared with the same type as the formal parameter.

This behaviours denote the body of Π where the formal gate parameter G_i is renamed into G'_i, the formal "**in**" (or "**inout**") parameter V_i is substituted with the value of E_i, and the formal signal parameter X_i is renamed into X'_i.

The evaluation of expressions in the list of actual parameters is done left-to-right.

Example 6.7.2
The process above is recursively instantiated after five time units and a rendezvous at its formal gate:

```
process CP [Tk] is
    wait (5); Tk; CP [Tk]
end process
```

■

The actual list of parameters may be specified in the "named" style, where the formal names of parameters appear. For example, the process instantiation using the "named" style for actual variable parameters has the form:

$$\Pi \ \texttt{[}G'_1, ..., G'_p\texttt{]} \ \texttt{(}V_1\texttt{=>}[\texttt{?}V'_1|E_1], ..., V_n\texttt{=>}[\texttt{?}V'_n|E_n]\texttt{)} \ \ \textbf{raises} \ \texttt{[}X'_1, ..., X'_m\texttt{]}$$

The application of Standard Lotos for realistic descriptions shows that the actual gate parameters are usually the same as the formal ones. Moreover, the gate parameter list is often very long (in some realistic example it has 20 elements). In this case, the process instantiation becomes very hard to write. Eliminating the actual gate parameters reduces sometimes the specification text of 20%.

To solve this problem, the keyword "**...**" may be used when the formal list of parameters uses the same names as the actual one.

Example 6.7.3
Although the number of gate parameters is not big for the process below, the use of "**...**" makes the specification shorter and clearer.

```
process Scheduler [Tk1, Tk2, Tk3]
        (d1, d2, d3: time)
is
   ...
    Scheduler [...] (d1-5,d2,d3)
   ...
    Scheduler [Tk1, Tk2, Tk3] (d1-5,d2,d3)
```

```
end process
```

■

The same convention may be used for the actual list of variables and signals.

6.8 Non-deterministic Behaviours

Non-determinism is an important feature of a specification language. Lotos NT offers several operators to express non-determinism. One of them is the non-deterministic assignment presented in Section 6.3. It is similar to the internal choice of CSP [Hoa85].

The operators presented in this section are external choice operators.

The interaction of real-time capabilities, non-deterministic choice (external or internal), and urgent actions (termination, internal, and signals) raises some problems which are discussed in [Sig99].

6.8.1 Operator "`[]`"

The unbracketed form of the choice operator has the form:

B_1 `[]` B_2

where B_1 and B_2 are behaviours. Informally, the "B_1 `[]` B_2" behaviour may execute either B_1 or B_2. In order to be able to control the variable initialization, B_1 and B_2 must initialize the same variables.

The start of the choice behaviour starts the evolution of both B_1 and B_2. They evolve in time with the same pace. If time is blocked on one side, it is also blocked for the whole process. The first action (*i.e.*, rendezvous, internal action, termination, or signaling) executed by one of B_1, B_2 (call it B_i) solves the choice in favor of B_i.

Example 6.8.1
The following behaviour models a coffee machine which, after receiving the payment (rendezvous at gate `MONEY`) may distribute tea, coffee, or chocolate (rendezvous at `TEA`, `COFFEE`, or `CHOCOLATE` gates):

```
MONEY;
    (    TEA
    []   COFFEE
    []   CHOCOLATE
    )
```

The non-deterministic choice means that the machine may accept any of these interactions. Its final behaviour is determined by the iteration with its *environment* (*i.e.*, the machine selection button). The three of the choices are active till one of the buttons is selected.

A timeout may be expressed using the non-deterministic choice and urgent actions. For example, the behaviour below will execute the internal action **i** if the interaction on gate `MONEY` waits more than 5 time units:

```
    MONEY
[]
    wait (5); i
```

The rendezvous on gate `MONEY` is available from the beginning of behaviour execution. After 5 time units, the second behaviour of the choice cannot evolve in time and must execute the internal urgent action **i**. So the choice is solved in favour of the second behaviour.

There are some cases where the choice is completely non-deterministic:

```
    signal X
[]
    signal Y
```

Both branches of the choice are urgent (cannot evolve in time). At the beginning of the behaviour, either the signal X or the signal Y is emitted. Since it is not possible to do a rendezvous on a signal (§ 6.9, p. 70), the behaviour is completely non-deterministic. ■

The operator "`[]`" is commutative, associative, and accepts "**stop**" as neutral element.

Note that the non-deterministic choice and the sequential composition are orthogonal operators. One may write:

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

which is not possible using the Lotos action prefix operator "`;`".

6.8.2 Operator choice over values

Lotos NT provides with the non-deterministic assignment a mean to choose a value of a domain. This choice is internal, *i.e.*, it does not depend on the environment. A derived behaviour is provided by choice over values which allows to choose a value of a finite enumerable domain.

Let B_0 be a behaviour which contains a use occurrence of variable V of the enumerable type T. Let B_N be the behaviour "$V\texttt{:=}N\texttt{;}B_0$" where N is a value of the domain of T. In order to express the behaviour:

$B_{N_1}\ \texttt{[]}\ ...\ \texttt{[]}\ B_{N_n}\ \texttt{[]}\ ...$

where $N_1, ..., N_n, ...$ are all the values of the domain of T, Lotos NT (like Lotos) allows the following shorthand notation:

choice $V\texttt{:}T$ `[]` B_0 **end choice**

The scope of V is the behaviour B_0.

The general form of the "**choice**" operator allows several iteration variables:

choice $\vec{V_1}\texttt{:}T_1, ..., \vec{V_n}\texttt{:}T_n$ `[]` B_0 **end choice**

where each variable of the list "$\vec{V_i}$" is a variable of type T_i.

The reception of a value (offer "`?`") may be expressed using the "**choice**" operator. For example:

$G\ \texttt{(?}V\texttt{)}\ \texttt{[}E\texttt{];}\ B_0$

where V is a variable of type T, is equivalent to:

choice $V\texttt{:}T$ `[]` $G\ \texttt{(!}V\texttt{)}\ \texttt{[}E\texttt{];}\ B_0$ **end choice**

In the current semantics of Lotos NT, the choice over values is a syntactic sugar of non-deterministic assignment. For example, the following behaviour:

choice $V_1\texttt{:}T_1\texttt{,}...\texttt{,}V_n\texttt{:}T_n$ `[]` B

may be translated using the non-deterministic assignment as follows:

var $V_1\texttt{:}T_1\texttt{,}...\texttt{,}V_n\texttt{:}T_n$ **in**
  V_1`:=` **any** T_1 `;` ... `;` V_n`:=` **any** T_n `;` B
end var

6.9 Concurrency

The behaviours presented till now are *sequential*; LOTOS NT (like LOTOS) provides means to put behaviours in *parallel* and to *synchronize* them. This section presents the parallel composition operators.

6.9.1 Synchronization operator

To express that two behaviours B_1 and B_2 evolve in parallel and synchronize on gates $G_1, ..., G_m$, and termination (δ), the following syntax is used:

B_1 `|[`$G_1, ..., G_m$`]|` B_2

Note that it is not possible to synchronize on the internal gate **i**.

The parallel behaviour forks its component behaviours, starting instantaneously all branches. When one branch offers a rendezvous at a gate G of the list $G_1, ..., G_m$, it must wait for the other branches to offer a rendezvous at G. If the rendezvous is possible, all the branches do an action *synchronously* on G carrying values exchanged by the rendezvous. The rendezvous at a gate different from $G_1, ..., G_m$ is done independently from other branches, in a *asynchronous* manner. The whole behaviour evolves in time when all its branches may evolve in time. The time passes with the same pace in all branches. The behaviour terminates when all branches terminate, waiting for the last one if some branches terminate earlier. In fact, the "δ" gate is always present—implicitly—in the synchronization list.

Variables can be shared among the parallel branches if they are read-only. If a branch writes a variable V, then no other branch can read nor write V.

Example 6.9.1
The coffee machine (example (§ 6.8.1, p. 68)) and a tea consumer doing her (his) choice after 5 time units may be put in parallel as follows:

```
MONEY;
    (    TEA
    []   COFFEE
    []   CHOCOLATE
    )
|[MONEY, TEA, COFFEE, CHOCOLATE]|
MONEY; wait (5); TEA
```

Since there are no time constraints on the rendezvous at the gate `TEA` from the part of the machine, this rendezvous will take place after 5 time units or more.

Note that the sequencing operator "`;`" has a higher precedence than the parallel operator, as explained in Section 6.12. ■

Note that synchronization between parallel behaviours may be done only using gates. There is no mean to synchronize parallel behaviours on signals. This constraint is largely discussed in [Sig99].

When the gate carries values, the rendezvous on the gate takes place iff the offers are compatible. Consider the two rendezvous below on the gate G:

G`[.`C^1`] (`V_1^1`=>`O_1^1`, ..., `$V_{n^1}^1$`=>`$O_{n^1}^1$`) [@`V^1`] [[`E^1`]]`
G`[.`C^2`] (`V_1^2`=>`O_1^2`, ..., `$V_{n^2}^2$`=>`$O_{n^2}^2$`) [@`V^2`] [[`E^2`]]`

They may synchronize if and only if the following four conditions are satisfied:

1. The tags C^1 and C^2 have the same name

2. The number of offers is the same $n^1 = n^2$
3. There exists a permutation ρ of indexes such that $V_i^1 = V_{\rho(i)}^2$ and O_i^1 and $O_{\rho(i)}^2$ match. The matching relation between different form of offers is synthesized in Table 6.1. Note that matching occurs only when offers have the same type, which is verified statically.

offer no 1	*offer no 2*	*matching condition*	*effect*	*name*
`!`E_1	`!`E_2	$\mathit{result}(E_1) = \mathit{result}(E_2)$	**null**	value matching
`?`V_1	`!`E_2	None	V_1`:=`E_2	value passing
`!`E_1	`?`V_2	None	V_2`:=`E_1	value passing
`?`V_1	`?`V_2	None	V_1`:=` **any** T`;`V_2`:=`V_1	value generation

Table 6.1: The matching relation between offers

4. The guards E^1 and E^2 evaluate to true.

Example 6.9.2
The behaviour below (see example (§ 6.4.2, p. 64)):

```
    Physical_Data.Header2 (source_id => ?src, dest_id => !3,
                           data_length => ?l, header_CRC => ?crc)
|[Physical_Data]|
    Physical_Data.Header2 (dest_id => ?dest, source_id => !0
                           data_length => !4, header_CRC => !1)
```

is equivalent to the behaviour

```
Physical_Data.Header2 (dest_id => !3, source_id => !0
                       data_length => !4, header_CRC => !1);
dest := 3;
src  := 0;
l    := 4;
crc  := 1
```

■

6.9.2 Full synchronization operator

A particular case of the synchronization operator is obtained when the list of synchronization gates is the list of all gates defined in the current context. The binary form of the *full synchronization* operator is the following:

B_1 `||` B_2

For example, the synchronization operator "`|[MONEY, TEA, COFFEE, CHOCOLATE]|`" of the example above may be simply written "`||`".

6.9.3 Interleaving operator

Another special case of the synchronization operator is obtained when the synchronization is done only on termination gate "δ". The binary form of the *interleaving* is the following:

B_1 `|||` B_2

Example 6.9.3
Suppose the existence of three independent consumers who interact with the coffee machine (example (§ 6.8.1, p. 68)). Each consumer asks a different drink. The consumers are modeled by interleaved behaviours using the "`|||`" operator. The interaction between the coffee machine and the consumers is a full synchronization.

```
MONEY;
    (    TEA
    []   COFFEE
    []   CHOCOLATE
    )
||
(
    MONEY; wait (5); TEA           (* first consumer *)
||| MONEY; wait (3); COFFEE        (* second consumer *)
||| MONEY; wait (1); CHOCOLATE     (* third consumer *)
)
```

■

6.9.4 Generalized parallel operator

The parallel composition operators play a crucial role in any non-trivial description of systems. In highly abstract (implementation-independent) descriptions, where the intended properties of the system are given as a set of mutually independent temporal constraints on the occurrences of certain events, parallelism is often used to express the logical conjunction of these properties. Such use of the parallel operator is known as the *constraint-oriented* specification style [VSS88]. In implementation-oriented descriptions, where the actual resources to be used in the implementation are identified and modeled (the *resource-oriented* specification style [VSS88]), parallel composition is used to describe the interconnection pattern of the implementation components.

So, a systems' description may be seen as a set of n processes communicating via a set of gates. This representation is also called a process-gate net of arity n [Bol90].

The necessary and sufficient conditions for a process-gate net to be translated into a LOTOS behaviour expression using its (binary) parallel composition operator are studied in [Bol90]. These results may be also applied to LOTOS NT. As a conclusion, [Bol90] claims the need for a more complex parallel composition operator, which should fill the gap between graphical and textual representation of process-gate nets.

The limits of previous parallel operators are essentially related to the fact that an expression is a linear object, while graphical representations consider multiple dimensions. The under-estimation of these limits may lead to ambiguous or complex textual descriptions.

To fill this gap, LOTOS NT provides a new and original operator, called the generalized parallel composition operator. It extends the network operator of CSP [Hoa85] in order to allow "m among n" synchronization. The syntax of this operator is:

par [**with** G_1[#m_1], ..., G_p[#m_p] **in**]
    $G_1^1, ..., G_{q_1}^1$ `->` B_1
  `||` ...
  `||`$G_1^n, ..., G_{q_n}^n$ `->` B_n
end par

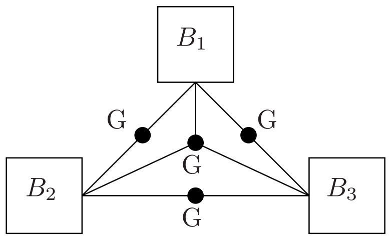

Figure 6.1: “2 among 3” and “3 among 3” synchronization on gate G

where $m_1, ..., m_p$ are natural numbers in the range $1, ..., n$. We define $\overline{G_0}$ to be the set $\{G_1, ..., G_p\}$, $\overline{G_i}$ to be the set $\{G_1^i, ..., G_{q_i}^i\}$ for $1 \leq i \leq n$ and we require that $\overline{G_0} \cap \overline{G_i} = \emptyset$ for all $1 \leq i \leq n$. Note that we do not require the gates $G_1, ..., G_p$ to be paiwise distinct.

The operator describes a set of behaviours $B_1, ..., B_n$ which are the processes of the net. Each behaviour B_i communicates with all others by the gates of the “interface” $\overline{G_i} = G_1^i, ..., G_{q_i}^i$ and by the gates of $\overline{G_0}$. The communication over a gate G_j^i of its interface should be synchronized with all other behaviours having the G_j^i in their interfaces. The communication over a gate $G_k \in \overline{G_0}$ should be synchronized with a number of behaviours equal to one of the *synchronization degree* declared for G_k. As for the classical parallel operator, the termination of behaviours is synchronous: all B_i's synchronize on the termination gate (“δ”).

Note that: (1) all gates should be already declared, and (2) B_i do not share variables (*i.e.*, a variable written by a behaviour cannot be read or written by another parallel behaviour).

The timed dynamic semantics respects the principle of ordinary parallel composition: all the processes evolve in time simultaneously.

The “**par**” operator can express both process interconnection and the degree of the interconnection in a simple manner. In the following examples we show how one may translate graphical representations into a textual one using this operator.

Example 6.9.4

In the process-gate net of Figure 6.1, behaviours B_i synchronize on gate G w.r.t. two patterns: “2 among 3” pattern and “3 among 3” pattern. In this case, the textual representation specifies the gate G in the “interface” list of each behaviour and the degrees 2 and 3 for G:

```
par with G#2, G#3 in
   B1 || B2 || B3
end par
```

Notice that the structure of the graphical representation is maintained. ■

Example 6.9.5

For the process-gate net of Figure 6.2, the textual representation specifies for each behaviour the gates through which it synchronizes. The degree of these gates is the default one, namely 3.

```
par
   G1, G3->B1 || G1, G4->B2
|| G1, G2, G3, G4->B3
|| G2, G3->B4 || G2, G4->B5
end par
```

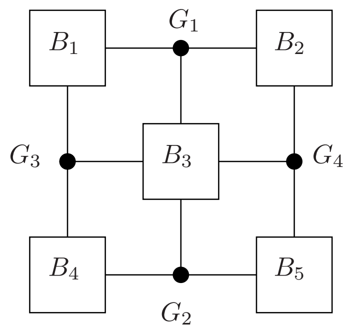

Figure 6.2: A two-dimensional process-gate net

■

6.9.5 Parallel over values operator

Similarly to the operator choice over value, LOTOS NT provides a mean to express parallel composition of behaviours iterating on a finite domain.

Let B_0 be a behaviour which contains a use occurrence of variable V of type T. Let "*op*" be one of the parallel composition operators "`|[...]|`", `||`", or "`|||`". Let B_N be the behaviour "V`:=`N`;`B_0" where N is a value of the domain of T. In order to express the behaviour:

B_{N_1} *op* ... *op* B_{N_n}

where values $N_1, ..., N_n$ are n values of the domain of T, LOTOS NT[1] allows the following shorthand notation:

par V:T **in** ES *op* B_0 **end par**

where ES is a list of expressions. The scope of V is the behaviour B_0.

Example 6.9.6
The behaviour below sends over the gate `G` all the working days of a week in any order:

```
par V:day_of_week in Monday, Tuesday, Wednesday, Thursday, Friday
    ||| G (!V); stop
end par
```

■

6.10 Hiding

LOTOS NT provides a mean to hide some gates of a given behaviour. The syntax of the operator is like in LOTOS, except that the (declared) gates are typed:

hide G_1:T_1, ..., G_n:T_n **in**
  B
end hide

[1]This operator is not a LOTOS operator.

The hide operator declares the gates $G_1, ..., G_n$ with their respective types in the behaviour B. The resulting behaviour substitutes all actions on the gates $G_1, ..., G_n$ by the internal action "**i**".

Example 6.10.1
A two places buffer may be expressed using two one place buffers which synchronize on a gate `mid`. The actions on gate `mid` are internal to the buffer.

```
hide mid: int in
    var x:int in
        input (?x); mid (!x)
        |[mid]|
        mid (?x); output (!x)
    end var
end hide
```

■

The substitution of the hidden gates by the internal urgent gate "**i**" allows to express that a synchronization should occur as soon as made possible by all the processes involved. For example the behaviour:

```
hide G in
     wait(1); G; B1
  || wait(2); G; B2
end hide
```

has the same semantics as:

```
wait(2); i;
hide G in
  B1 || B2
end hide
```

The hidden G occurs after 2 time units, which is as soon as both processes can perform G.

The behaviour:

```
hide G in
  G@?t[t > 3]; B
end hide
```

has two possible semantics depending on whether the type time is discrete or dense. If `time` is a synonym for natural number (discrete time), the behaviour has the same semantics as:

```
wait(4); t:=4; i;
hide G in B end hide
```

because 4 is the smallest natural number strictly greater than 3. On the other hand, if time is a synonym for rational number (dense time), the behaviour has the same semantics as:

```
wait(3); block
```

The reason why this process time stops after 3 time units without even performing the hidden G is because there is no smallest rational (or earliest time) strictly greater than 3.

The urgency of hidden actions is inherited from ET-LOTOS. It is largely discussed in [LL97]. Having to hide synchronizations to make them occur as soon as possible is sometimes criticized, because there are cases where one would still like to observe those gates. The problem here lies in the interpretation of the word "observation". Observing requires interaction, and interaction may lead to interference. Clearly, we would like to show the interaction to the environment without allowing it to interfere. There is a nice solution to this problem. It suffices to raise an exception (signal) immediately after the occurrence of the hidden interaction.

Example 6.10.2
Consider the two places buffer example (§ 6.10.1, p. 75), and suppose that one wants to signal the passing of the value from one place to another, as soon as it takes place. For this, a special monitoring behaviour is added. It synchronizes on `mid` and then raises the `Transfer` signal:

```
hide mid: int in
    var x:int in
        input (?x); mid (!x)
        |[mid]|
        mid (?x); output (!x)
        |[mid]|
        mid (?x); signal Transfer    (* the monitor *)
    end var
end hide
```

Since the signaling is urgent, `Transfer` will be signalled the same time as the internal action resulted from synchronization on gate `mid`. ■

6.11 Renaming

The "**rename**" operator has the following form:

> **rename**
> *ren-clause*$_1$
> . . .
> *ren-clause*$_n$
> **in**
> B
> **end rename**

where $n \geq 0$, and the renaming clauses are given by the following grammar:

ren-clause	::=	**exception** X $[P]$ $:T$ **is** X' $[E]$	*renaming exceptions*
	|	**gate** G $[P]$ $:T$ **is** $G'[.C]$ $(V_1$=>$O_1, ..., V_p$=>$O_p)$	*renaming gates*

where $m, p \geq 0$. Note that the named form of rendezvous may be substituted with a positional one. Also, the terms on the right of the keyword "**is**" are similar to signaling and communication behaviours, except that for exceptions, the keyword "**raise**" is not specified. For this reason, these terms must satisfy the constraints of the corresponding behaviour expressions (see Sections 6.4 and 6.6): the signals X' and the gates G' must be defined in the current context, and their parameters (if exist) must be of compatible types.

The "**rename**" operator declares the signals X and the gates G in the behaviour B. Their type is T. The pattern P matches the values carried by X or G; its variables are visible only in E or $O_1, ..., O_n$. The gate G cannot be the internal gate **i**.

The "**rename**" behaviour is equivalent to the behaviour B where, at the execution (*i.e.*, in the LTS), the signal X carrying a value matching the corresponding pattern P is substituted with the signal X' carrying the result of E, and where communication on gate G carrying values matching the pattern P is substituted with communication on gate G' with the corresponding offers. This semantics for renaming corresponds to a post-relabelling operator like in Timed CSP [Sch93].

Renaming an observable action into another observable action may be much more powerful than one might think at first, because it allows one to do more than just renaming gate names. For example, it can be used to change the structure of events occurring at a gate (adding or removing attributes), or to merge or split gates.

The simplest form of renaming just renames one gate to another:

```
rename gate G x: int is G' !x
in ...
end rename
```

This form of renaming is so common that a shorthand for it is provided:

```
rename gate G : int is G'
in ...
end rename
```

The renaming allows the removing of a field from a gate:

```
rename gate G (x: int, y: bool) : Record2 is G !x
in ...
end rename
```

where the type `Record2` has the following definition:

```
type Record2 is record x: int, y: bool end type
```

Note that `G` has the type `Record2` in the body of the "**rename**" behaviour, and it has the type `int` in the context of the "**rename**" behaviour.

A new field may be added to the gate:

```
rename gate G x: int is G (x => !x, y => !true)
in ...
end rename
```

where `G` is already declared with type `Record2` in the context of the "**rename**" behaviour.

Two gates `G1` and `G2` can be merged into a single gate `G`:

```
rename gate G1 x: int is  G (x => !x, y => !true)
       gate G2 x: int is  G (x => !x, y => !false)
in ...
end rename
```

A gate `G` may be split into gates `G1` and `G2`:

```
rename gate G (x: int, true)  : Record2 is G1 !x
       gate G (x: int, false) : Record2 is G2 !x
in ...
end rename
```

Signals may be renamed in a similar way.

6.12 Unambiguous Behaviours

The priority degree of binary operators of LOTOS NT control part is given in table 6.2. The operators with a higher priority degree have lower priority (or give precedence). For example B_1 `[ ]` B_2`;` B_3

is parsed B_1 `[ ]` (B_1`;` B_3) because the sequential composition has a smaller priority degree than the choice operator.

Priority	*Operators*
0.	basic behaviours, actions, assignments, signaling, bracketed operators
1.	`;`
2.	`||`, `|[ ... ]|`, `|||`
3.	`[]`

Table 6.2: The precedence of LOTOS NT behaviour operators.

This precedence can be forced by using parenthesized behaviours:

`(`B`)`

Parenthesis may be also used for esthetic considerations. Its semantics is the same as the semantics B.

All parallel binary operators are right associative.

Chapter 7

Modules

This chapter presents the structure of a LOTOS NT specification and a number of related concepts: modularity, equational specification, genericity.

The modularity is an important concept for specifying "in the large". It is well known from practical specification that, especially when large specifications are developed, one wishes to encapsulate one or more types, operations, or processes so that they can be regarded as a single unit. It is also common experience that a major part of the specification effort lies in deciding what are these logical units of the specification. Finally, it is desirable that manipulation of such units can be expressed in the specification language itself, so that it becomes subject to automatic checking. In conclusion, the module system shall provide mean for structuring, abstraction, and re-usability.

One of the goals of LOTOS NT is to develop a modularization system, which should allow export and import, hiding, and genericity. This goal is justified by the limited form of modularity offered by LOTOS. The modules in LOTOS encapsulate data types and operations, but not processes. Moreover, this mechanism does not support abstraction: every object declared in a module is exported outside. These deficiencies make LOTOS difficult to use, and cause problems for users and tool implementors alike. A critical evaluation of LOTOS data types from the user point of view can be found, for instance, in [Mun91].

NOTE: Since 1988, there have been several proposals for enhancing LOTOS modules. An overview of those proposals is given in [Que96, Section 3].

Our proposal is based on the ACTONE style of modules (called types), and extend it in two directions:

- The specification of processes is allowed inside the module units. This allows new processes to be generated as types (by renaming and instantiation).

- The genericity feature of ACTONE is cleaned in order to offer a better control of the formal (generic) parameters of a specification.

- In order to avoid cumbersome renaming, a one-level dot notation is provided. The identifiers may be "long", that is one may specify explicitly the name of the definition module for a type, constructor, function, and process identifier.

  The one-level depth of the dotted notation implies that importation is, like in ACTONE but unlike SML or IDL, non generative (*i.e.*, the imported objects do not become local objects of the module importing them. Introducing generative importation complicates the identifier analysis and the static semantics checking.

7.1 Generalities

The module system concerns both parts of the language: the data part and the control part, so process declarations are allowed as well as type and function declarations. The LOTOS NT module language allows to define a set of related objects — types, functions, and processes, and to control the visibility of these objects at each point of the description.

There are two important concepts: the module *interface* which declares the visible objects of a module and what is necessary to use them (e.g., function profiles), and the definition *module* in which the actual description of objects (visible or not visible) is given. The interface is an outside view of a module, and represents a "contract" between the author of the module and its users, while the module contains the inner mechanisms which should remain hidden to the module user.

Three features improve this clean separation between the module structure and the module description:

- The visible part of a module may be seen through several interfaces provided that these interfaces respect the declaration given by the module.

- Types and other objects in an interface may be declared as being *opaque* (or abstract), which means that their structure is not know by the user; only functions and processes provided in the corresponding implementation module may operate in an opaque type; functions and processes are always opaque.

- A module may be *parameterized* ("generic") by a collection of objects to adapt general behaviours to particular situations.

Each declaration of an object O inside an interface I declares the *scope interface* of O as being I. Each definition of an object O inside a module M declares the *scope module* of O as being M. The scope interfaces and modules are associated with each object identifier occurrence. This association is called *extended* object identifier. It allows the solving of names conflicts by using the "dotted" notation for identifiers: "*scp*.*id*" where *scp* is the scope (module or interface) of the identifier *id*. This notation has one level of depth because it is not allowed to nest modules or interfaces.

The structured set of objects of an interface (or module) is called the *signature* of the interface (or module). The set is structured into three classes: the class of types, the class of functions, and the class of processes. Each class binds the extended identifier of the object with the *declaration* (profile) of the object.

In an interface (or module) signature, the objects of the same class must have different extended identifiers (*i.e.*, either different module definitions, or different names) except for functions, where overloading is allowed.

In conclusion, a specification in modular LOTOS NT is a set of *interface* declarations (see Section 7.2), *module* declarations (see Section 7.3), *generic modules* declarations (see Section 7.4), and a unique entry point declaration called *specification*[1] (see Section 7.5).

[1]This will change in future versions of LOTOS NT.

7.2 Interfaces

7.2.1 Interface declaration

An interface describes the external (visible) part of the objects — types, functions, and processes — described inside a module. The parameters of a generic module (§ 7.4, p. 86), and the objects imported from other modules, may enter in this collection.

The visible part of a type is its name and maybe its definition (implementation). If the definition is given (§ 4.1, p. 27), all the operations applicable to that type (pattern-matching, field selection, etc.) are available. Otherwise, the type is opaque. An opaque type declaration has the form:

type T

In this case, no operation is implicitly defined for this type, except (non-)equality functions (`==`, `!=`). The only functions defined on such an opaque type are those declared in the interface. Opaque types are very useful to implement *abstract data types*.

The visible part of a process is its complete header:

process Π [`[`G_1`:`T_1`, ...,` G_p`:`T_p`]`][`(`[A_1] V_1`:`T_1`, ...,` [A_n] V_n`:`T_n`)`]
  [**raises** `[`X_1`:`T_1`, ...,` X_m`:`T_m`]`]

where A_i may be "**in**", "**out**", or "**inout**". The types of formal parameters must be either predefined, imported, or declared by the interface. The body of a process never appears inside an interface, i.e processes are always opaque. NOTE: A proposal to describe the process interface by giving equivalence relations or logic formulas is given in [GM96, Sig99].

The visible part of a function is its complete header:

function F `(`[A_1] V_1`:`T_1`, ...,` [A_n] V_n`:`T_n`)` [`:`T]
  [**raises** `[`X_1[`:`T_1]`, ...,` X_m[`:`T_m]`]`]

where A_i may be "**in**", "**out**", or "**inout**". The same constraints for the types used in the profile must be satisfied. As for processes, the body of a function never appears inside an interface. However, it is possible to specify the axioms that the function satisfies, using *equations* (§ 7.2.3, p. 83).

An interface declaration has the general form:

interface *int-id* [**import** $IE_1, ..., IE_n$] **is**
  IB
end interface

where *int-id* is the interface identifier, $IE, IE_1, ..., IE_n$ are *interface expressions*. $IE_1, ..., IE_n$ are called *imported interfaces*, and IB is called the *interface body*. The importation cannot be circular, as explained in Section 7.6.

7.2.2 Interface expression

An interface expression may be:

- An interface declaration: This interface expression can appear only in an interface body.

  Example 7.2.1
  The interface of a `Monoid` module (*i.e.*, a domain with an associative and commutative operation "`+`", whose neutral elements is "`0`") is:

  ```
  interface Monoid is
  ```

```
    type Monoid
    function 0: Monoid
    function + (x: Monoid, y: Monoid) : Monoid
end interface
```

■

- An interface identifier *int-id*′: In an importation clause, this means that the objects of *int-id*′ are imported in *int-id*, but their declaration interface is *int-id*′. In the interface body, this means that the objects of *int-id* have the same form as those of *int-id*′, but their declaration interface is *int-id*.

  Example 7.2.2
  A monomorphic list is a monoid with a single type of elements, denoted by `E`:

```
interface List import Monoid is
    type E
    function inj (x: E) : Monoid
end interface
```

  The function `inj` is a bijection from the element type domain to the list domain. ■

- An interface renaming:

  `[`*int-id*′ **renaming** `(`*ren-list*`)]`

  where *ren-list* are explicit renaming of *int-id*′ objects. Renamings have the form:

ren-list	::=	[**types** *ren*]	*type*
		[**opns** *ren*]	*operations*
		[**procs** *ren*]	*processes*
ren	::=	id'_1 `:=` $id_1, \ldots, id'_n$ `:=` id_n	

  where the unprimed identifiers are the new names (of *int-id*), and the primed identifiers are the old names (of *int-id*′). The renaming clauses declare an isomorphism from the *int-id* names to the *int-id*′ names, which is the identity for the names of *int-id*′ which are not specified in the renaming list. This isomorphism is uniformly applied to all the profiles of objects of the interface.

  Example 7.2.3
  The interface for a `String` monoid module may be obtained by renaming the `Monoid` interface:

```
interface String is
    [ Monoid renaming (types Monoid := string
                       opns  0      := empty  ) ]
end interface
```

  This declaration is equivalent to the following one:

```
interface String is
    type string
    function empty : string
    function + (x: string, y: string) : string
end interface
```

Note that for "+", the renaming isomorphism is an identity, but its profile is changed according to the renaming of the Monoid type. ■

- An interface body, possibly renamed:

  [*IB* [**renaming** (*ren-list*)]]

  This form is useful to give directly the profile of the objects, without explicitly declaring an interface.

All objects visible by an interface, including the imported objects, are visible from outside and may be imported in other interfaces. The importation is transitive through interfaces. Also, an interface may be transitively imported several times. This is not a conflictual situation, the interface being imported one time (the "diamond" scheme).

It is not allowed to have multiple declarations for an identifier belonging to the same class of objects, except for functions if the overloading constraints are respected. For example, if the interface *int-id* declares a type T, it cannot declare another type having the same name.

7.2.3 Equation declaration

Interfaces may contain an axiomatic specification of functions. These descriptions are given using ActOne equations for compatibility with Lotos data language. However, the equations are just like (type checked) comments. It is ensured that (like in Extended ML) the equations can be commented out without affecting the semantics of the module.

The equation declarations have the form:

eqns
  forall $V_1 : T_1, ..., V_n : T_n$
  ofsort T
    [E'_1=>] E_1 = E'_1;
    ...
    [E'_p=>] E_p = E'_p;
end eqns

where $V_1, ..., V_n$ are variables used by the expressions E_i, E'_i, and T is the type of the expressions E_i, E'_i. The meaning of an equation "E'_i=> E_i = E'_i" is that the terms E_i and E'_i denote the same value in the domain of T if the premise E'_i is true.

Example 7.2.4
In the Monoid interface, the function "+" may be characterized by the following equations:

```
eqns
    forall x, y, z: M
    ofsort M
        x + 0 = x ;
        0 + x = x ;
        (x + y) + z = x + (y + z) ;
end eqns
```

which express that the operation "+" is associative and has 0 as neutral element. ■

7.3 Modules

The Lotos NT modules contain the description (implementation) of a set of related types, functions, and processes. The visibility of these objects is controlled using interfaces.

7.3.1 Module declaration

The declaration of a Lotos NT module has the form:

> **module** *mod-id* [:IE] [**import** $ME_1, ..., ME_n$] **is**
> MB
> **end module**

where *mod-id* is the name of the module, IE is an interface expression (§ 7.2, p. 81), $ME_1, ..., ME_n$ are *module expressions* (§ 7.3.2, p. 85), and MB is the *module body*.

The interface IE controls[†] the visibility of objects defined in the module *mod-id*: all the objects declared in IE are visible. Note that IE is self contained (§ 7.2, p. 81). By default, the interface of the module is the list of objects imported by the importing clause, and the list of objects declared by the module body.

The import clause makes the objects declared by the module expressions $ME_1, ..., ME_n$ visible inside the module.

The body of a module defines the objects of the module — types, functions, and processes. Types may be defined as shown in Section (§ 4.1, p. 27). Functions are defined using function definitions (§ 5.5.1, p. 54), and processes are defined as indicated in Section (§ 6.7.1, p. 66). Another mean to define the module objects is using the module expression (§ 7.3.2, p. 85), *i.e.*, renaming and generic module instantiation.

The module definitions must be self-contained: all the objects used must be either imported by the importation clause, or defined in the current module. The reference to an object which is not visible in the module is forbidden. Recursive definitions of types, processes, and functions are allowed.

Example 7.3.1
The specification of a data-flow process is:

```
module DataFlow is
    type Record2 is
        record x: int, y: int
    end type
    process Flow [Input: Record2, Output: int] is
    var x: int, y: int
    in
        loop
          Input (?x, ?y);
          Output !(x+y)
        end loop
    end var
    end process
end module
```

The default interface for the `DataFlow` is:

```
interface DataFlow is
    type Record2
    process Flow [Input: Record2, Output: int]
end interface
```

Note that the standard module of the predefined types is included by default, an explicit importation clause is not needed. ■

Example 7.3.2
The following module defines the one point domain:

```
module OnePoint is
    type M is zero end type
    function 0: M is return zero end function
    function infix + (x: M, Y: M) : M is
        case x is
            zero -> case y is zero -> return zero end case
        end case
    end function
end module
```

■

Note that modules are not nested.

7.3.2 Module expression

A module expression may be:

- A module identifier *mod-id′*: If it appears in the importation clause, it corresponds to the importation of all visible objects of the module. If it appears as the body of a module *mod-id*, the objects of *mod-id* are the same definitions as the objects of *mod-id′*, but their definition module is *mod-id* (instead of *mod-id′*).
- A module identifier *mod-id′* restricted by an interface *int-id′*, noted "*mod-id′* : *int-id′*". If it appears in the importation clause, it corresponds to the importation of the objects M' declared in *int-id′*. If it appears as the body of a module, the objects of *mod-id* are the objects declared in *int-id′* with the definition given by *mod-id′*, but the definition module is *mod-id*.
- A renaming module expression:

  mod-id′ [: I'] **renaming** (*ren-list*)

  where the renaming clauses *ren-list* are described in (§ 7.2.2, p. 81). The renaming clauses declare an isomorphism from the *mod-id* names to the *mod-id′* names, which is identity for the names of *mod-id′* which are not specified in the renaming list.
- A generic module instantiation (§ 7.4.2, p. 87).

Example 7.3.3
The `Monoid` module is constructed by renaming the `OnePoint` module as follows:

```
module Monoid is
    OnePoint renaming (types M := Monoid)
end module
```

■

7.3.3 External modules

External module declaration is provided for interfacing with other specification or implementation languages. An external module declaration has the form:

> **external module** $M : IE$ **is**
> *file-id*
> **end module**

The interface of the module is compulsory because it gives the names and the profiles of the objects implemented by the module.

7.4 Generic Modules

Genericity is a useful tool for building specifications and for code reuse. It is provided in Lotos NT by the generic modules.

7.4.1 Generic module declaration

The declaration of a generic module has the form:

> **generic** *gen-id* (*mod-id*$_1 : IE_1, ..., $*mod-id*$_n : IE_n$) [$: IE$]
> [**import** $ME_1, ..., ME_p$] **is**
> MB
> **end generic**

where *gen-id* is the name of the generic module, *mod-id*$_1, ...,$ *mod-id*$_n$ are module identifiers, $IE, IE_1, ..., IE_n$ are interface expressions (§ 7.2.2, p. 81), $ME_1, ..., ME_n$ are module expressions (§ 7.3.2, p. 85), and MB is the *module body* (§ 7.3, p. 84).

The module identifiers *mod-id*$_1, ...,$ *mod-id*$_n$ are also called the parameters of the generic module. The visible objects of *mod-id*$_i$ are those declared by the interface expression IE_i. They may be used by the objects defined in the generic module body.

Example 7.4.1
The interface below specify modules implementing monomorphic lists:

```
interface List import Monoid is
    type E
    function inj (x: E) : M
end interface
```

A generic module implementing monomorphic lists (example (§ 7.2.2, p. 82)) is parameterized by the type of elements of the list:

```
interface EqType is
    type E
end interface

generic GenericList (Eq: EqType) : List is
    type M is nil, cons (e: E, l: M) end type
    function 0 : M is return nil end function
    function infix + (x: M, y: M) : M is
        case x is var z: E, zl: M in
            nil -> return y
        |   cons (z, zl) -> return cons (z, zl + y)
        end case
    end function
    function inj (x: E) : M is
        return cons (x, nil ())
    end function
end generic
```

■

7.4.2 Generic module instantiation

A generic module instantiation replaces the formal parameters of the generic module with the actual parameters which give the implementation of the formal objects. Its simplest form is:

gen-id ($ME_1, ..., ME_n$)

where the module expressions $ME_1, ..., ME_n$ define the objects declared by the interface expressions $IE_1, ..., IE_n$ (*i.e.*, the parameters of the generic module *mod-id*). So, the actual parameters shall match the interface specified for the formal parameter in all objects. For instance, the constructors shall be instantiated with constructors in order to flatten correctly the pattern-matching expressions.

Example 7.4.2
The module below defines the type `E` as being a one point domain:

```
module Element : EqType is
    OnePoint renaming (types M := E)
end module
```

The interface of the module is the `EqType` interface, so the module `Element` may be used to instantiate the `GenericList` module:

```
module OnePointList : List is
    GenericList (Element)
end module
```

Note that the type `M` declared by the module `OnePoint` must be renamed in `E` before instantiation, in order to match the interface `EqType`. For this reason, the module `Element` is declared. ■

Declaring a module for each module instantiation may become cumbersome. For this, a simpler form form of generic module instantiation is provided:

gen-id ($ren_1, ..., ren_p$)

where $ren_1, ..., ren_p$ are renaming of form "*id* `=>` *id'*" where *id* is the identifier of the formal objects declared by the generic module parameters, and *id'* is the identifier of the actual parameter. The actual objects must be visible inside the instantiation context.

Example 7.4.3
Using this syntax, the previous example may be written:

```
module OnePointList : List import OnePoint is
    GenericList (types E := OnePoint::M)
end module
```

Only the objects declared in the interface `List` are visible. Those imported from the module `OnePoint` are not visible. The type `E` has the same definition as the type `M`.

To obtain a list of integers, one has to declare:

```
GenericList (types E := int)
```

because the type `int` is visible in each point of the specification. ■

The generic module instantiation creates new objects. These objects have as definition module the module where the instantiation is done. For example, the module `OnePointList` has as (visible) objects the type `E`, the function `0`, and the functions "`+`" and "`inj`"; all these objects have as definition module `OnePointList`.

Generic modules may be partially instantiated or used to build other generic modules.

Example 7.4.4
For example, consider the following generic module implementing stacks of generic elements of type `E`:

```
generic GenericStacks (Eq: EqType) is
    type Stack is empty, push (e: E, s: Stack) end type
    function pop (s: Stack) : Stack raises EMPTY_STACK:none is
        case s is
            var s1: Stack in
            empty -> raise EMPTY_STACK
        |   push (any, s1) -> return s1
        end case
    end function
    function top (s: Stack) : E raises EMPTY_STACK:none is
        case s is var x: E in
            empty -> raise EMPTY_STACK
        |   push (x, any) -> return x
        end case
    end function
end generic
```

In order to obtain stacks of (generic) list, one shall declare:

```
generic StacksOfList (Eq: EqType) import GenericList (Eq) is
    GenericStacks (types E := M)
end generic
```

In the resulting generic module, the formal (generic) element `Eq` is used to instantiate the `GenericList` module and then the `GenericStack` module is instantiated using the list type `M` instead of the formal element `E`. ■

7.5 Specification

The entry point on a LOTOS NT description is the *specification* declaration. It defines a list of imported modules, a list of gates, a list of signals (exceptions), and the body of the specification. The body of the specification may be either a behaviour or an expression.

> **specification** Σ [**import** $ME_1, ..., ME_k$] **is**
> [**gates** $G_1 : T_1, ..., G_p : T_p$]
> [**exceptions** $X_1 : T_1, ..., X_m : T_m$]
> (**behaviour** B | **value** I)
> **end specification**

Note the similarity between specification definition and function or process definition. For this reason the gates and the exceptions declared in a specification are also called the parameters of the specification. This corresponds to the interface of the system modeled by the specification.

The object used by the behaviour B or by the expression E are the parameters of the specification and all other objects imported by the specification.

Note that the types used must be already declared.

7.6 Importation Rules

If some interface (or module) A imports some objects from interface (or module) B, then A is said to be *dependent* on B. To prevent any problem with circular definitions[2], it is required that this "dependency" relation does not contain cycle.

[2] Such as an object X of A defined in terms of an object Y of B and vice versa.

Appendix A

Syntax Summary

This chapter presents the full concrete grammar (syntax) of the language. The notations used are those presented in Chapter 2. The lexical structure of the language is defined in Chapter 3.

A.1 Syntax of the module part

Equation declaration

eqn	::=	$[E\ \{,E\}\Rightarrow]\quad E = E$	*simple equation*	(eqns1)
eqn-list	::=	**ofsort** T *eqn* {; *eqn*}	*list of equations*	(eqns2)
eqns	::=	[**forall** VL] *eqn-list* {*eqn-list*}	*equations*	(eqns3)

Extended identifiers:

id	::=	$T \mid F \mid C \mid \Pi$	*simple identifiers*	(eid1)
uid	::=	*mod-id* | *int-id* | *gen-id*	*unit identifier*	(eid2)
eid	::=	*id*	*simple extended identifier*	(eid3)
	|	*uid* :: *id*	*extended identifier*	(eid4)

Renaming list:

R	::=	*eid*:=*eid* {,*eid*:=*eid*}	*renaming*	(RL1)
RL	::=	[**types** R]	*type renaming*	(RL2)
		[**opns** R]	*operation renaming*	
		[**procs** R]	*process renaming*	

Interface body:

IB	::=	**type** T [**with** F {,F}]	*abstract type*	(IB1)
	|	**type** T **is** TD [**with** F {,F}] **end type**	*complete type*	(IB2)
	|	**function** F [(VFL)] [:T] [**raises** [XL]]	*function interface*	(IB3)
	|	**process** Π [[GL]] [(VFL)] [**raises** [XL]]	*process interface*	(IB4)
	|	**eqns** *eqns* **end eqns**	*equations*	(IB5)
	|	IB IB	*sequence*	(IB6)

Interface expressions:

IE	::=	*int-id* [**renaming** (RL)]	*identifier*	(IE1)
	|	[IB]	*explicit interface*	(IE2)

Module body:

MB	::=	D {D}	*declarations*	(MB1)
	|	ME	*module expressions*	(MB2)

Module expressions:

ME	::=	*mod-id*	*module identifier*	(ME1)
	|	*gen-id* (MS)	*instantiation*	(ME2)
	|	*gen-id* (RL)	*instantiation with renaming*	(ME3)
	|	ME:IE	*constraint*	(ME4)
	|	ME **renaming** (RL)	*renaming*	(ME5)

Sequence of module expressions:

MS	::=		*empty*	(MS1)
	|	*mod-id* $\Rightarrow$ ME {,*mod-id* $\Rightarrow$ ME}	*disjoint union*	(MS2)
	|	ME {,ME}	*list*	(MS3)

Unit declaration:

UD	::=	**interface** *int-id* [**import** IE {,IE}] **is** IB **end interface**	*interface*	(UD1)

	| **module** *mod-id* [:IE] [**import** ME {,ME}] **is** MB **end module**	*simple module*	(UD2)
	| **external module** *mod-id* :IE [**import** ME {,ME}] **is** *file* **end external**	*external module*	(UD3)
	| **generic** *gen-id* (*mod-id*:IE {,*mod-id*:IE}) [:IE] [**import** ME {,ME}] **is** MB **end generic**	*generic module*	(UD4)
	| **specification** *spec-id* [**import** ME {,ME}] **is** [**gates** G_1:T_1, ..., G_p:T_p] [**exceptions** X_1:T_1, ..., X_m:T_m] (**behaviour** B | **value** I) **end specification**	*specification*	(UD5)

Description:

descr	::=	UD {UD}	Lotos NT *description*	(descr1)

Declarations:

D	::=	**type** T **is** TD [**with** F {,F}] **end type**	*type*	(D1)
	|	**function** F [(VFL)] [:T] [**raises** [XL]] **is** I **end function**	*function*	(D2)
	|	**process** Π [[GL]] [(VFL)] [**raises** [XL]] **is** B **end process**	*process*	(D3)

A.2 Syntax of the data part

Attributes of parameters:

A	::=	[**in**]	*input formal parameter*	(A1)
	|	**out**	*output formal parameter*	(A2)

	$\mid$	**inout**	*input/output formal parameter*	(A3)

List of variable:

$\overline{V}$	$::=$	$V\ \{,V\}$	*list of variable identifiers*	(VL1)
VL	$::=$	$\overline{V}:T\ \{,\overline{V}:T\}$	*list of variables*	(VL2)

VFL	$::=$	$A\ \overline{V}:T\ \{,A\ \overline{V}:T\}$	*formal parameter list*	(VFL1)

List of exceptions:

$\overline{X}$	$::=$	$X\ \{,X\}$	*list of exception identifiers*	(XL1)
XL	$::=$	$\overline{X}:T\ \{,\overline{X}:T\}$	*list of exceptions*	(XL2)

Type definition:

TD	$::=$	$C\ [(VL)]\ \{,\ C\ [(VL)]\}$	*constructed type*	(TD1)
	$\mid$	**enum** $C\ \{,C\}$	*enumeration*	(TD2)
	$\mid$	**record** VL	*structure*	(TD3)
	$\mid$	**set of** T	*set*	(TD4)
	$\mid$	**list of** T	*list*	(TD5)
	$\mid$	**array** $[T\ \{,T\}]$ **of** T	*array*	(TD6)
	$\mid$	$'\{' V:T,\ E\ '\}'$	*subtype*	(TD7)

Sequence of expressions:

ES	$::=$	$[\ldots]$	*empty or wildcard*	(ES1)
	$\mid$	$V \Rightarrow E\ \{,V \Rightarrow E\}\ [,\ldots]$	*disjoint union*	(ES2)
	$\mid$	$E\ \{,E\}$	*list*	(ES3)

Sequence of exceptions:

XS	$::=$	$[\ldots]$	*empty or wildcard*	(XS1)
	$\mid$	$X \Rightarrow X\ \{,X \Rightarrow X\}\ [,\ldots]$	*disjoint union*	(XS2)
	$\mid$	$X\ \{,X\}$	*list*	(XS3)

Expressions:

E	::=	K	*primitive constant*	(E1)
	|	V	*variable*	(E2)
	|	C [`(`ES`)`]	*constructor application*	(E3)
	|	$E\ C\ E$	*infix constructor application*	(E4)
	|	`[` E {`,`E} `]`	*list or set construction*	(E5)
	|	**array** T `[` E {`,`E} `]`	*array construction*	(E6)
	|	F [`(`ES`)`]	*function call*	(E7)
	|	$E\ F\ E$	*infix function call*	(E8)
	|	V `[`$E_1, ..., En$`]`	*array access*	(E9)
	|	E`.`V	*field selection*	(E10)
	|	E`.`'{' ES '}'	*field update*	(E11)
	|	E **of** T	*type coercion*	(E12)
	|	E **raises** `[`$\overline{X}$`]`	*raise exception expression*	(E13)
	|	`(`E`)`	*parenthesized expression*	(E14)

The precedence of operators appearing in expressions is given on table A.2.

Priority	*Operations*
0.	**raises**
1.	`-` unary
2.	**of**, `.` field selection and update
3.	`and`, `andthen`, `/`, `%`, `*`, `**`
4.	`or`, `orelse`, `xor`, `+`, `-`
5.	`==`, `!=`, `<`, `<=`, `>=`, `>`, `iff`, `implies`

All the binary arithmetic operators are left associative.

Sequences of patterns:

PS	::=	[**...**]	*empty or wildcard*	(PS1)
	|	$V \Rightarrow P$ {`,`$V \Rightarrow P$} [`,`**...**]	*disjoint union*	(PS2)
	|	P {`,`P}	*list*	(PS3)

Patterns:

P	::=	V	*variable*	(P1)
	|	**any**	*wildcard*	(P2)
	|	V **as** P	*aliasing*	(P3)
	|	K	*constant*	(P4)
	|	C [`(`PS`)`]	*constructed pattern*	(P5)
	|	$P\ C\ P$	*constructed pattern infixed*	(P6)
	|	`[` P {`,`P} `]`	*list pattern*	(P7)

	$\mid$	P **of** T	*explicit typing*	(P8)

Match statements:

	$\mid$	$P\ \{\texttt{\vert}\ P\}\ [\texttt{[}E\texttt{]}]\ \texttt{->}\ I$	*composed*	(IM1)
	$\mid$	$IM\ \texttt{\vert}\ IM$	*list*	(IM2)

Actual value parameters:

VE	$::=$	E	*actual parameter "in"*	(VE1)
	$\mid$	$\texttt{?}V$	*actual parameter "out" and "inout"*	(VE2)

VS	$::=$	$[\texttt{...}]$	*empty or wildcard*	(VS1)
	$\mid$	$V \Rightarrow VE\ \{\texttt{,}V \Rightarrow VE\}\ [\texttt{,...}]$	*disjoint union*	(VS2)
	$\mid$	$VE\ \{\texttt{,}VE\}$	*list*	(VS3)

Statements:

I	$::=$	**return** E	*value return*	(I 1)
	$\mid$	**null**	*termination*	(I 2)
	$\mid$	$V\texttt{:=}E$	*assignment*	(I 3)
	$\mid$	$I\ \texttt{;}\ I$	*sequential*	(I 4)
	$\mid$	**var** $\overline{V}\texttt{:}T[\texttt{:=}E]\ \{\texttt{,}\overline{V}\texttt{:}T[\texttt{:=}E]\}$ **in** I **end var**	*variable declaration*	(I 5)
	$\mid$	**case** $E\ [\texttt{:}T]$ **is** [**var** VL **in**] IM **end case**	*case statement*	(I 6)
	$\mid$	**if** E **then** I { **elsif** E **then** I } [**else** I] **end if**	*conditional statement*	(I 7)
	$\mid$	**eval** $[V\texttt{:=}]\ F\ [\texttt{(}VS\texttt{)}]$ [**raises** $\texttt{[}XS\texttt{]}$]	*procedure call*	(I 8)
	$\mid$	**loop** I **end loop**	*forever loop*	(I 9)
	$\mid$	**loop** X **in** I **end loop**	*breakable loop*	(I 10)
	$\mid$	**while** E **do**	*while loop*	(I 11)

	I		
	end while		
|	**for** I **while** E **by** I **do**	*for loop*	(I 12)
	I		
	end for		
|	**break** X	*break loop*	(I 13)
|	**raise** X $[E]$	*raise exception*	(I 14)
|	**trap** $\{$**exception** $X[:T]$ **is** $I\}$ **in**	*trapping exceptions*	(I 15)
	I		
	end trap		

A.3 Syntax of the behaviour part

Gate lists:

$\overline{G}$	::=	$G\ \{,G\}$	*list of gate identifiers*	(GL1)
GL	::=	$\overline{G}{:}T\ \{,\overline{G}{:}T\}$	*list of gate declarations*	(GL2)

Sequences of gates:

GS	::=	$[$**...**$]$	*empty or wildcard*	(GS1)
	|	$G \Rightarrow G\ \{,G \Rightarrow G\}\ [,$**...**$]$	*disjoint union*	(GS2)
	|	$G\ \{,G\}$	*list*	(GS3)

Sequences of offers:

OS	::=		*empty*	(OS1)
	|	V `=>` $O\ \{,V$ `=>` $O\}$	*disjoint union*	(OS2)
	|	$O\ \{,O\}$	*list*	(OS3)

Offers:

O	::=	`!`E	*send a value*	(O1)
	|	`?`P	*receive a value*	(O2)
	|	‘.‘C `(`OS`)`	*constructed offer*	(O3)
	|	`(` OS `)`	*record*	(O4)

Match behaviour:

BM	::=	P [`[`E`]`] `->` B	*simple*	(BM1)
	|	P {`|` P} [`[`E`]`] `->` B	*composed*	(BM2)
	|	BM `|` BM	*list*	(BM3)

Behaviours:

B	::=	**block**	*time block*	(B1)
	|	**wait** `(`E`)`	*delay*	(B2)
	|	**null**	*successful termination*	(B3)
	|	**stop**	*inaction*	(B4)
	|	V`:=`E	*assignment*	(B5)
	|	**eval** [V`:=`] F [`(`VS`)`] [**raises** `[`XS`]`]	*function call*	(B6)
	|	G [O] [`@`P] [`[`E`]`]	*communication action*	(B7)
	|	**i**	*internal action*	(B8)
	|	B `;` B	*sequencing*	(B9)
	|	**var** $\overline{V}$`:`T[`:=`E] {`,`$\overline{V}$`:`T[`:=`E]} **in** B **end var**	*variable declaration*	(B10)
	|	Π [`[`GS`]`] [`(`VS`)`] [**raises** `[`XS`]`]	*process instantiation*	(B11)
	|	**case** E [`:`T] **is** [**var** VL **in** BM **end case**	*case*	(B12)
	|	**if** E **then** B { **elsif** E **then** B} [**else** B] **end if**	*conditional*	(B13)
	|	**signal** X [E]	*signaling*	(B14)
	|	**raise** X [E]	*raising exception*	(B15)
	|	**break** X	*breaking iteration*	(B16)
	|	**trap** {**exception** X [V`:`T] **is** B} **in** B **end trap**	*exception trapping*	(B17)
	|	**loop** I **end loop**	*loop forever*	(B18)
	|	**loop** X **in** B **end loop**	*named loop*	(B19)
	|	**while** E **do** B	*while loop*	(B20)

	end while		
|	**for** I **while** E **by** I **do**	*for loop*	(B21)
	B		
	end for		
|	**rename**	*renaming*	(B22)
	$\{$(**gate** G $[P]$`:`T **is** G $[O]$) | (**exception** X $[P]$`:`T **is** X $[E]$)$\}$		
	in B		
	end rename		
|	B `[]` B	*choice*	(B23)
|	V`:=`**any** T [`[`E`]`]	*non deterministic assignment*	(B24)
|	**choice** VL `[]` B **end choice**	*choice over values*	(B25)
|	B *op-par* B	*parallel*	(B26)
|	**par** [**with** G[`#`n]$\{$`,`G[`#`n]$\}$ **in**]	*general parallel*	(B27)
	$\overline{G}$`->`$B\{$`||`$\overline{G}$`->`$B\}$		
	end par		
|	**par** V`:`T **in** $E\{$`,`$E\}$	*parallel over values*	(B28)
	op-par B		
	end par		
|	**hide** GL **in**	*gate hiding*	(B29)
	B		
	end hide		
|	`(`B`)`	*parenthesized behaviour*	(B30)

op-par	::=	`|||`	*interleaving*	(op-par1)
	|	`||`	*fully synchronization*	(op-par2)
	|	`|[`$\overline{G}$`]|`	*synchronization*	(op-par3)

The level of priority for the (base) operators is given by the Table A.1.

Priority	*Operators*
0.	basic behaviours, actions, assignments, signaling, bracketed operators
1.	`;`
2.	`||`, `|[ ... ]|`, `|||`
3.	`[]`

Table A.1: The precedence of LOTOS NT behaviour operators.

All parallel operators are right associative.

Bibliography

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[Bol90] T. Bolognesi. A Graphical Composition Theorem for Networks of Lotos Processes. In IEEE Computer Society, editor, *Proceedings of the 10th International Conference on Distributed Computing Systems, Washington, USA*, pages 88–95. IEEE, May 1990.

[Cd95] J.P. Courtiat and R.C. de Oliveira. A Reachability Analysis of RT-LOTOS Specifications. Technical Report 95159, LAAS, May 1995.

[CGM$^+$96] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, October 1996. Full version available as INRIA Research Report RR-2958.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to Algebraic Specifications Based on the Language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[GH93] Hubert Garavel and René-Pierre Hautbois. An Experiment with the Formal Description in LOTOS of the Airbus A340 Flight Warning Computer. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *First AMAST International Workshop on Real-Time Systems (Iowa City, Iowa, USA)*, November 1993.

[GM96] Hubert Garavel and Radu Mateescu. French-Romanian Proposal for Capture of Requirements and Expression of Properties in E-LOTOS Modules. Rapport SPECTRE 96-04, VERIMAG, Grenoble, May 1996. Input document [KC4] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Kansas City, Missouri, USA, May, 12–21, 1996.

[Gut77] J. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ISO89a] ISO/IEC. LOTOS Description of the Session Protocol. Technical Report 9572, International Organization for Standardization — Open Systems Interconnection, Genève, 1989.

[ISO89b] ISO/IEC. LOTOS Description of the Session Service. Technical Report 9571, International Organization for Standardization — Open Systems Interconnection, Genève, 1989.

[ISO92a] ISO/IEC. Distributed Transaction Processing — Part 3: Protocol Specification. International Standard 10026-3, International Organization for Standardization — Information Technology — Open Systems Interconnection, Genève, 1992.

[ISO92b] ISO/IEC. Formal Description of ISO 8072 in LOTOS. Technical Report 10023, International Organization for Standardization — Telecommunications and Information Exchange between Systems, Genève, 1992.

[ISO92c] ISO/IEC. Formal Description of ISO 8073 (Classes 0, 1, 2, 3) in LOTOS. Technical Report 10024, International Organization for Standardization — Telecommunications and Information Exchange between Systems, Genève, 1992.

[ISO95a] ISO/IEC. LOTOS Description of the CCR Protocol. Technical Report 11590, International Organization for Standardization — Open Systems Interconnection, Genève, 1995.

[ISO95b] ISO/IEC. LOTOS Description of the CCR Service. Technical Report 11589, International Organization for Standardization — Open Systems Interconnection, Genève, 1995.

[JSV93] A. Jeffrey, S. Schneider, and F. Vaandrager. A Comparison of Additivity Axioms in Timed Transition Systems. Technical Report cs1193, COGS, University of Sussex, 1993.

[LL95] R. Lai and A. Lo. An Analysis of the ISO FTAM Basic File Protocol Specified in LOTOS. *Australian Computer Journal*, 27(1):1–7, February 1995.

[LL97] Luc Léonard and Guy Leduc. An Introduction to ET-LOTOS for the Description of Time-sensitive Systems. *Computer Networks and ISDN Systems*, 29:271–292, September 1997.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mun91] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[NS94] Xavier Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.

[Pec94] Charles Pecheur. A proposal for data types for E-LOTOS. Technical Report, University of Liège, October 1994. Annex H of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[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)*. IFIP, North-Holland, June 1992.

[Que96] 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.

[Sch88] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming*, 11:133–159, 1988.

[Sch93] S. Schneider. Rigorous Specification of Real-time Systems. In *Proceedings of 3rd International Conference on Algebraic Methodology and Software Technology AMAST'93, Twente, The Netherlands*, June 1993.

[Sig99] Mihaela Sighireanu. *Contribution à la définition et à l'implémentation du langage "Extended LOTOS"*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1999.

[VSS88] C. Vissers, G. Scollo, and M. van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In S. Aggarwal and K. Sabnani, editors, *Proceedings of the 8th International Workshop on Protocol Specification, Testing and Verification (Atlantic City, NJ, USA)*, pages 189–204. IFIP, North-Holland, 1988.

[WWF87] D. Watt, B. Wichmann, and W. Findlay. *ADA Language and Methodology*. Prentice-Hall, 1987.

Index

δ, 64
..., 67
any
    assignment, 62
    offer, 64
array, 37
behaviour, 89
block, 59
break
    behaviour, 61
    statement, 50
call, 63
case
    behaviour, 60
    statement, 48
choice, 69
else, 49, 61
elsif, 49, 61
enum, 33
eqns, 83
exceptions, 89
exception, 53, 62
exit, 62
external, 86
false, 28, 30
forall, 83
for
    behaviour, 60, 61
    statement, 52
function, 54, 81
gates, 89
gate, 76
generic, 86
hide, 74
if
    behaviour, 60, 61
    statement, 49
import
    generic module, 86
    interface, 81
    module, 84
    specification, 89
inout, 54
interface, 81
in, 47, 53, 54, 60, 62, 66, 72, 74, 76
is, 48, 54, 60, 66
i, 64
list, 36
loop
    behaviour, 60, 61
    breakable, 50, 52, 54, 61
    forever, 50, 61
    statement, 50
module, 84, 86
noexit, 66
null, 47
    behaviour, 59
ofsort, 83
of, 43
out, 54, 66
par, 72
process, 66, 81
raises, 54, 66
raise
    behaviour, 60, 62
    statement, 52, 53
record, 34
rename
    behaviour, 76
renaming, 82, 83, 85
set, 35
signal, 65, 76
specification, 89
stop, 59
then, 49, 61
trap
    behaviour, 60, 62
    statement, 53
true, 28, 30
type, 81
value, 89
var

behaviour, 60
statement, 47
wait, 59
while
behaviour, 60, 61
statement, 51
†, 3, 13
`TRAIAN_INIT`, 54
`[]` (choice), 68

action
gate, 63
internal, 64
signal, 65
termination, 65
assignment
behaviour, 62
non-deterministic, 62
statement, 47

behaviour
basic, 59
regular, 60
BNF (Backus-Naum Form), 16

call
by name, 41
by position, 41
choice, 68
over values, 69
communication
guard, 63
offers, 64
pure, 64
receive offer, 64
send offer, 64
time, 63
conditional
behaviour, 60
statement, 48
constructor
application, 41

data
carrier, 17
domain, 17

equation, 83
exception
raising, 52, 62
trapping, 62
expression, 40
parenthesized, 43

function, 54
call, 56, 63
named, 56
positional, 56
definition, 54
interface, 81

gate, 63
synchronization degree, 73
generic module
declaration, 86
instantiation, 85, 87

handler, 53

identifier
extended, 80
importation, 89
interface, 80, 81
declaration, 81
expression, 81
renaming, 82, 83
iterative
behaviour, 61
statement, 50

labelled transition system
internal event, 17
labels, 17
structured, 17

module, 80
body, 84
declaration, 84
expression, 84, 85
external, 86
renaming, 85

non-determinism, 68

parallel, 70
interleaving, 71
with full synchronization, 71
with synchronization, 70
parameter
inout, 55, 56
in, 55, 56
out, 55, 56

actual, 56
exception, 55
formal, 55
value, 55
pattern, 44
in case, 48
pragma, 37
pragmas
operation, 38
type, 37
process
definition, 66
instantiation, 67
interface, 81
process-gate net, 72

renaming, 82, 85
behaviour, 76
rendezvous, 63
return
statement, 47

scope, 80
selection, 42
sequential
behaviour, 65
statement, 47
signal, 62, 65
raising, 65
signature
interface, 80
module, 80
specification, 89
statement, 45
syntax
concrete, 17

token
`CHAR`, 24
`DECIMAL_NUMBER`, 23
`FLOATING_POINT`, 23
`HEX_NUMBER`, 23
`OCTAL_NUMBER`, 23
`STRING`, 24
type, 27
array, 37
bool, 30
char, 31
constructor, 27
declaration, 27
enumerable, 34
enumerated, 33
finite, 34
finite enumerable (see scalar), 34
int, 30
interface (opaque), 81
list, 35
nat, 30
none, 32, 53
real, 31
record, 34
scalar, 34
set, 35
string, 32
subtype, 37
typing
explicit, 43

update, 43

value, 41
primitive constants, 39
variable, 41
declaration, 47, 60