

AS/NZS ISO/IEC 15437:2003  
ISO/IEC 15437:2001  
Reconfirmed 2014

Australian/New Zealand Standard™

**Information technology—Open  
distributed processing—Enhancements  
to LOTOS (E-LOTOS)**



**Standards Australia**



**STANDARDS  
Documents**

Document Sharing Hub

### **AS/NZS ISO/IEC 15437:2003**

This Joint Australian/New Zealand Standard was prepared by Joint Technical Committee IT-015, Software Engineering. It was approved on behalf of the Council of Standards Australia on 6 May 2003 and on behalf of the Council of Standards New Zealand on 22 April 2003. It was published on 4 June 2003.

---

The following are represented on Committee IT-015:

Australian Computer Society  
Australian Information Industry Association  
Australian Society for Technical Communication (NSW)  
Australian Software Metrics Association  
Griffith University  
New Zealand Organisation for Quality  
Quality Society of Australasia  
Software Engineering Australia (Qld)  
Software Quality Association (ACT)  
Software Quality Association (NSW)  
Software Verification Research Centre  
Sydney SPIN Group (Software Process Improvement Network)  
Systems Engineering Society of Australia  
University of New South Wales  
University of South Australia  
University of Technology, Sydney

---

### **Keeping Standards up-to-date**

Standards are living documents which reflect progress in science, technology and systems. To maintain their currency, all Standards are periodically reviewed, and new editions are published. Between editions, amendments may be issued. Standards may also be withdrawn. It is important that readers assure themselves they are using a current Standard, which should include any amendments which may have been published since the Standard was purchased.

Detailed information about joint Australian/New Zealand Standards can be found by visiting the Standards Australia web site at [www.standards.com.au](http://www.standards.com.au) or Standards New Zealand web site at [www.standards.co.nz](http://www.standards.co.nz) and looking up the relevant Standard in the on-line catalogue.

Alternatively, both organizations publish an annual printed Catalogue with full details of all current Standards. For more frequent listings or notification of revisions, amendments and withdrawals, Standards Australia and Standards New Zealand offer a number of update options. For information about these services, users should contact their respective national Standards organization.

We also welcome suggestions for improvement in our Standards, and especially encourage readers to notify us immediately of any apparent inaccuracies or ambiguities. Please address your comments to the Chief Executive of either Standards Australia International or Standards New Zealand at the address shown on the back cover.

---

*This Standard was issued in draft form for comment as DR 03092.*

STANDARDS AUSTRALIA/STANDARDS NEW ZEALAND

---

**RECONFIRMATION**

**OF**

**AS/NZS ISO/IEC 15437:2003**

**Information technology—Open distributed processing—Enhancements to LOTOS  
(E-LOTOS)**

---

**RECONFIRMATION NOTICE**

Technical Committee IT-015 has reviewed the content of this publication and in accordance with Standards Australia procedures for reconfirmation, it has been determined that the publication is still valid and does not require change.

Certain documents referenced in the publication may have been amended since the original date of publication. Users are advised to ensure that they are using the latest versions of such documents as appropriate, unless advised otherwise in this Reconfirmation Notice.

Approved for reconfirmation in accordance with Standards Australia procedures for reconfirmation on 17 March 2014.

The following are represented on Technical Committee IT-015:

Australian Computer Society  
Australian Society for Technical Communication (NSW)  
Charles Sturt University  
Department of Defence (Australia)  
Griffith University  
La Trobe University  
National Association of Testing Authorities Australia  
National ICT Australia  
NSW Business Chamber  
Quantitative Enterprise Software Performance  
Systems Engineering Society of Australia  
University of Technology, Sydney

---

## NOTES

# Australian/New Zealand Standard™

## Information technology—Open distributed processing—Enhancements to LOTOS (E-LOTOS)

First published as AS/NZS ISO/IEC 15437:2003.

### **COPYRIGHT**

© Standards Australia/Standards New Zealand

All rights are reserved. No part of this work may be reproduced or copied in any form or by any means, electronic or mechanical, including photocopying, without the written permission of the publisher.

Jointly published by Standards Australia International Ltd, GPO Box 5420, Sydney, NSW 2001, and Standards New Zealand, Private Bag 2439, Wellington 6020

ISBN 0 7337 5284 5

## PREFACE

This Standard was prepared by the Joint Standards Australia/Standards New Zealand Committee IT-015, Software Engineering.

This Standard is identical with, and has been reproduced from ISO/IEC 15437:2001, *Information technology—Enhancements to LOTOS (E-LOTOS)*.

The objective of this Standard is to define the syntax and semantics of the enhanced LOTOS language, named E-LOTOS. E-LOTOS is used for the formal description of the behavioural aspects of distributed and concurrent systems in general and in the area of open distributed processing in particular.

As this Standard is reproduced from an international standard, the following applies:

- (a) Its number appears on the cover and title page while the international standard number appears only on the cover.
- (b) In the source text ‘this International Standard’ should read ‘this Australian/New Zealand Standard’.
- (c) A full point substitutes for a comma when referring to a decimal marker.

References to International Standards should be replaced by references to Australian or Australian/New Zealand Standards, as follows:

### *Reference to International Standard*

ISO

8807 Information processing systems—  
Open systems interconnection—  
LOTOS—A formal description  
technique based on the temporal  
ordering of observational behaviour

### *Australian/New Zealand Standard*

AS/NZS

4199 Information processing systems—  
Open systems interconnection—  
LOTOS—A formal description  
technique based on the temporal  
ordering of observational behaviour

## CONTENTS

|                                                       | <i>Page</i> |
|-------------------------------------------------------|-------------|
| 1 Scope . . . . .                                     | 1           |
| 2 Conformance . . . . .                               | 1           |
| 3 Normative reference . . . . .                       | 1           |
| 4 Terms, definitions and notation . . . . .           | 1           |
| 5 E-LOTOS grammar . . . . .                           | 2           |
| 5.1 Lexical Structure . . . . .                       | 2           |
| 5.1.1 Character set . . . . .                         | 2           |
| 5.1.2 Comments and separators . . . . .               | 3           |
| 5.1.3 Identifiers . . . . .                           | 3           |
| 5.1.4 Reserved words . . . . .                        | 3           |
| 5.1.5 Identifiers classes . . . . .                   | 4           |
| 5.1.6 Non-terminals classes . . . . .                 | 4           |
| 5.2 Syntax of the language . . . . .                  | 5           |
| 5.2.1 Specification . . . . .                         | 5           |
| 5.2.2 Top-level declaration . . . . .                 | 6           |
| 5.2.3 Module body . . . . .                           | 6           |
| 5.2.4 Module expression . . . . .                     | 6           |
| 5.2.5 Module formal parameters . . . . .              | 6           |
| 5.2.6 Interface expressions . . . . .                 | 6           |
| 5.2.7 Record module expression . . . . .              | 6           |
| 5.2.8 Interface body . . . . .                        | 7           |
| 5.2.9 Formal parameter list . . . . .                 | 7           |
| 5.2.10 Renaming/Instantiation . . . . .               | 7           |
| 5.2.11 Equations declaration . . . . .                | 7           |
| 5.2.12 Simple equations declaration . . . . .         | 8           |
| 5.2.13 Declarations . . . . .                         | 8           |
| 5.2.14 Expressions . . . . .                          | 8           |
| 5.2.15 Expression Atoms . . . . .                     | 8           |
| 5.2.16 Behaviour expressions . . . . .                | 9           |
| 5.2.17 Disabling behaviour expression . . . . .       | 10          |
| 5.2.18 Synchronization behaviour expression . . . . . | 10          |

|        |                                                      |    |
|--------|------------------------------------------------------|----|
| 5.2.19 | Concurrency behaviour expression . . . . .           | 10 |
| 5.2.20 | Selection behaviour expression . . . . .             | 10 |
| 5.2.21 | Suspend/Resume behaviour expression . . . . .        | 10 |
| 5.2.22 | Interleaving behaviour expression . . . . .          | 10 |
| 5.2.23 | Behaviour term . . . . .                             | 10 |
| 5.2.24 | Behaviour atom . . . . .                             | 11 |
| 5.2.25 | Type expressions . . . . .                           | 12 |
| 5.2.26 | Record type expressions . . . . .                    | 12 |
| 5.2.27 | Value expressions . . . . .                          | 12 |
| 5.2.28 | Record value expressions . . . . .                   | 13 |
| 5.2.29 | Patterns . . . . .                                   | 13 |
| 5.2.30 | Gate parameter list . . . . .                        | 13 |
| 5.2.31 | Actual parameter list . . . . .                      | 13 |
| 5.2.32 | Exception parameter list . . . . .                   | 13 |
| 5.2.33 | Record patterns . . . . .                            | 14 |
| 5.2.34 | Behaviour pattern matching . . . . .                 | 14 |
| 5.2.35 | Expression pattern matching . . . . .                | 14 |
| 5.2.36 | In parameter list . . . . .                          | 14 |
| 5.2.37 | In parameter . . . . .                               | 14 |
| 6      | E-LOTOS abstract syntax . . . . .                    | 14 |
| 6.1    | Overview . . . . .                                   | 14 |
| 6.1.1  | Syntactic sugar . . . . .                            | 15 |
| 6.1.2  | Abstract syntax . . . . .                            | 15 |
| 6.2    | Concrete to abstract syntactic translation . . . . . | 16 |
| 6.2.1  | Interface body . . . . .                             | 16 |
| 6.2.2  | Formal parameter list . . . . .                      | 18 |
| 6.2.3  | Declarations . . . . .                               | 18 |
| 6.2.4  | Expressions . . . . .                                | 21 |
| 6.2.5  | Expression atoms . . . . .                           | 21 |
| 6.2.6  | Behaviour expressions . . . . .                      | 24 |
| 6.2.7  | Interleaving behaviour expression . . . . .          | 24 |
| 6.2.8  | Behaviour atom . . . . .                             | 25 |
| 6.2.9  | Type expressions . . . . .                           | 31 |
| 6.2.10 | Record type expressions . . . . .                    | 32 |
| 6.2.11 | Record value expressions . . . . .                   | 32 |
| 6.2.12 | Gate parameter list . . . . .                        | 33 |
| 6.2.13 | Actual parameter list . . . . .                      | 33 |
| 6.2.14 | Exception parameter list . . . . .                   | 33 |
| 6.2.15 | Record patterns . . . . .                            | 33 |
| 6.2.16 | In parameter list . . . . .                          | 34 |
| 6.2.17 | In parameter . . . . .                               | 34 |
| 7      | E-LOTOS semantics . . . . .                          | 35 |
| 7.1    | Overview . . . . .                                   | 35 |
| 7.2    | Static Semantics . . . . .                           | 36 |
| 7.2.1  | Static semantic objects for Base . . . . .           | 36 |
| 7.2.2  | Judgements on static semantics for Base . . . . .    | 37 |
| 7.2.3  | Extended identifiers . . . . .                       | 38 |
| 7.2.4  | Static semantic objects for Modules . . . . .        | 39 |
| 7.2.5  | Judgements on static semantics for Modules . . . . . | 39 |
| 7.2.6  | Cycle freedom . . . . .                              | 40 |
| 7.2.7  | Context morphism . . . . .                           | 40 |
| 7.2.8  | Realization . . . . .                                | 40 |

|        |                                                                        |    |
|--------|------------------------------------------------------------------------|----|
| 7.2.9  | Interface Instantiation . . . . .                                      | 40 |
| 7.2.10 | Interface Matching . . . . .                                           | 41 |
| 7.2.11 | Renaming/Instantiation . . . . .                                       | 41 |
| 7.3    | Untimed dynamic semantics . . . . .                                    | 44 |
| 7.3.1  | Untimed dynamic semantic objects for Base . . . . .                    | 44 |
| 7.3.2  | Judgements on untimed dynamic semantics for Base . . . . .             | 45 |
| 7.3.3  | Dynamic semantic objects for Modules . . . . .                         | 46 |
| 7.3.4  | Judgements on untimed dynamic semantics for Modules . . . . .          | 47 |
| 7.3.5  | Environment morphism . . . . .                                         | 48 |
| 7.3.6  | Signature Instantiation . . . . .                                      | 48 |
| 7.3.7  | Renaming/Instantiation . . . . .                                       | 48 |
| 7.4    | Timed dynamic semantics . . . . .                                      | 48 |
| 7.4.1  | Judgements on timed dynamic semantics . . . . .                        | 49 |
| 7.5    | Write-many variables: the value substitution operator . . . . .        | 49 |
| 8      | The E-LOTOS modules . . . . .                                          | 52 |
| 8.1    | Specification . . . . .                                                | 52 |
| 8.1.1  | Specification . . . . .                                                | 52 |
| 8.2    | Top-level declaration . . . . .                                        | 53 |
| 8.2.1  | Module not constrained by an interface . . . . .                       | 53 |
| 8.2.2  | Module constrained by an interface . . . . .                           | 53 |
| 8.2.3  | Generic module not constrained by an interface . . . . .               | 54 |
| 8.2.4  | Generic module constrained by an interface . . . . .                   | 55 |
| 8.2.5  | Interface declaration . . . . .                                        | 55 |
| 8.2.6  | Sequential top declaration . . . . .                                   | 56 |
| 8.3    | Module body . . . . .                                                  | 56 |
| 8.3.1  | Block declaration . . . . .                                            | 56 |
| 8.3.2  | Module Expression . . . . .                                            | 57 |
| 8.4    | Module expression . . . . .                                            | 57 |
| 8.4.1  | Module aliasing not constrained by an interface . . . . .              | 57 |
| 8.4.2  | Module aliasing constrained by an interface . . . . .                  | 58 |
| 8.4.3  | Generic module actualization not constrained by an interface . . . . . | 58 |
| 8.4.4  | Generic module actualization constrained by an interface . . . . .     | 59 |
| 8.4.5  | Generic module renaming/instantiation . . . . .                        | 59 |
| 8.5    | Module formal parameters . . . . .                                     | 60 |
| 8.5.1  | Single . . . . .                                                       | 60 |
| 8.5.2  | Disjoint union . . . . .                                               | 60 |
| 8.6    | Interface expressions . . . . .                                        | 61 |
| 8.6.1  | Interface identifier . . . . .                                         | 61 |
| 8.6.2  | Simple renaming . . . . .                                              | 61 |
| 8.6.3  | Explicit body . . . . .                                                | 61 |
| 8.7    | Interface body . . . . .                                               | 62 |
| 8.7.1  | Type hiding the implementation . . . . .                               | 62 |
| 8.7.2  | Type synonym . . . . .                                                 | 62 |
| 8.7.3  | Constructed type . . . . .                                             | 62 |
| 8.7.4  | Named record type . . . . .                                            | 63 |
| 8.7.5  | Process declaration . . . . .                                          | 63 |
| 8.7.6  | Equations . . . . .                                                    | 64 |
| 8.7.7  | Sequential declaration . . . . .                                       | 64 |
| 8.8    | Record module expression . . . . .                                     | 64 |
| 8.8.1  | Single . . . . .                                                       | 64 |
| 8.8.2  | Disjoint union . . . . .                                               | 65 |
| 8.8.3  | Renaming tuple . . . . .                                               | 65 |

|        |                                                 |    |
|--------|-------------------------------------------------|----|
| 8.9    | Equation declarations . . . . .                 | 65 |
| 8.9.1  | Equations declaration . . . . .                 | 65 |
| 8.9.2  | Sequential . . . . .                            | 66 |
| 8.10   | Simple equation declaration . . . . .           | 66 |
| 8.11   | Declarations . . . . .                          | 66 |
| 8.11.1 | Type synonym . . . . .                          | 66 |
| 8.11.2 | Type declaration . . . . .                      | 66 |
| 8.11.3 | Named record type . . . . .                     | 67 |
| 8.11.4 | Process declaration . . . . .                   | 67 |
| 8.11.5 | Sequential declarations . . . . .               | 68 |
| 9      | The E-LOTOS base language . . . . .             | 69 |
| 9.1    | Introduction . . . . .                          | 69 |
| 9.2    | Behaviours . . . . .                            | 69 |
| 9.2.1  | Disabling behaviour expression . . . . .        | 69 |
| 9.2.2  | Synchronization behaviour expression . . . . .  | 70 |
| 9.2.3  | Concurrency behaviour expression . . . . .      | 71 |
| 9.2.4  | Selection behaviour expression . . . . .        | 73 |
| 9.2.5  | Suspend/Resume behaviour expression . . . . .   | 73 |
| 9.2.6  | Sequential composition . . . . .                | 74 |
| 9.2.7  | Action . . . . .                                | 75 |
| 9.2.8  | Internal action . . . . .                       | 76 |
| 9.2.9  | Successful termination without values . . . . . | 76 |
| 9.2.10 | Successful termination . . . . .                | 76 |
| 9.2.11 | Inaction . . . . .                              | 77 |
| 9.2.12 | Time block . . . . .                            | 77 |
| 9.2.13 | Delay . . . . .                                 | 77 |
| 9.2.14 | Assignment . . . . .                            | 78 |
| 9.2.15 | Nondeterministic Assignment . . . . .           | 78 |
| 9.2.16 | Choice over values . . . . .                    | 79 |
| 9.2.17 | Trap . . . . .                                  | 80 |
| 9.2.18 | General parallel . . . . .                      | 81 |
| 9.2.19 | Parallel over values . . . . .                  | 83 |
| 9.2.20 | Variable declaration . . . . .                  | 84 |
| 9.2.21 | Gate hiding . . . . .                           | 84 |
| 9.2.22 | Renaming . . . . .                              | 85 |
| 9.2.23 | Process instantiation . . . . .                 | 87 |
| 9.2.24 | <b>loop</b> iteration . . . . .                 | 87 |
| 9.2.25 | Case . . . . .                                  | 88 |
| 9.2.26 | Case with tuples . . . . .                      | 88 |
| 9.2.27 | Signalling . . . . .                            | 89 |
| 9.3    | Type expressions . . . . .                      | 90 |
| 9.3.1  | Type identifier . . . . .                       | 90 |
| 9.3.2  | Empty type . . . . .                            | 90 |
| 9.3.3  | Universal type . . . . .                        | 90 |
| 9.3.4  | Record type . . . . .                           | 90 |
| 9.4    | Record type expressions . . . . .               | 91 |
| 9.4.1  | Singleton record . . . . .                      | 91 |
| 9.4.2  | Universal record . . . . .                      | 91 |
| 9.4.3  | Record disjoint union . . . . .                 | 91 |
| 9.4.4  | Empty record . . . . .                          | 92 |
| 9.5    | Value expressions . . . . .                     | 92 |
| 9.5.1  | Primitive constants . . . . .                   | 92 |

|        |                                                       |     |
|--------|-------------------------------------------------------|-----|
| 9.5.2  | Variables . . . . .                                   | 92  |
| 9.5.3  | Record values . . . . .                               | 92  |
| 9.5.4  | Constructor application . . . . .                     | 93  |
| 9.6    | Record value expressions . . . . .                    | 93  |
| 9.6.1  | Singleton record . . . . .                            | 93  |
| 9.6.2  | Record disjoint union . . . . .                       | 93  |
| 9.6.3  | Empty record . . . . .                                | 93  |
| 9.7    | Patterns . . . . .                                    | 94  |
| 9.7.1  | Record pattern . . . . .                              | 94  |
| 9.7.2  | Wildcard . . . . .                                    | 94  |
| 9.7.3  | Variable binding . . . . .                            | 94  |
| 9.7.4  | Expression pattern . . . . .                          | 95  |
| 9.7.5  | Constructor application . . . . .                     | 95  |
| 9.7.6  | Explicit typing . . . . .                             | 96  |
| 9.8    | Record patterns . . . . .                             | 96  |
| 9.8.1  | Singleton record pattern . . . . .                    | 97  |
| 9.8.2  | Record wildcard . . . . .                             | 97  |
| 9.8.3  | Record match . . . . .                                | 97  |
| 9.8.4  | Record disjoint union . . . . .                       | 98  |
| 9.8.5  | Empty record pattern . . . . .                        | 98  |
| 9.9    | Record of variables . . . . .                         | 98  |
| 9.9.1  | Singleton record variable . . . . .                   | 98  |
| 9.9.2  | Record disjoint union . . . . .                       | 98  |
| 9.10   | Behaviour pattern-matching . . . . .                  | 99  |
| 9.10.1 | Single match . . . . .                                | 99  |
| 9.10.2 | Multiple match . . . . .                              | 100 |
| 10     | Predefined library . . . . .                          | 100 |
| 10.1   | Booleans . . . . .                                    | 101 |
| 10.2   | Natural Numbers . . . . .                             | 103 |
| 10.3   | Integral Numbers . . . . .                            | 106 |
| 10.4   | Rational Numbers . . . . .                            | 109 |
| 10.5   | Floating Point Numbers . . . . .                      | 112 |
| 10.6   | Characters . . . . .                                  | 113 |
| 10.7   | Strings . . . . .                                     | 114 |
| 10.8   | Enumerated Type Scheme . . . . .                      | 114 |
| 10.9   | Record Type Scheme . . . . .                          | 116 |
| 10.10  | Set Type Scheme . . . . .                             | 117 |
| 10.11  | List Type Scheme . . . . .                            | 120 |
| A      | Tutorial . . . . .                                    | 122 |
| A.1    | The base language . . . . .                           | 122 |
| A.1.1  | Basic concepts . . . . .                              | 124 |
| A.2    | The module language . . . . .                         | 145 |
| A.2.1  | Basic concepts . . . . .                              | 148 |
| A.3    | An E-LOTOS specification of the ODP trader . . . . .  | 156 |
| A.3.1  | Introduction . . . . .                                | 156 |
| A.3.2  | An overview of the ODP Trader . . . . .               | 156 |
| A.3.3  | E-LOTOS Specification of the trader . . . . .         | 158 |
| A.3.4  | The complete specification . . . . .                  | 161 |
| B      | Guidelines for LOTOS to E-LOTOS translation . . . . . | 179 |
| B.1    | Introduction . . . . .                                | 179 |
| B.1.1  | Specification and process definition . . . . .        | 180 |
| B.1.2  | Basic LOTOS . . . . .                                 | 180 |

|       |                      |     |
|-------|----------------------|-----|
| B.1.3 | Data Types . . . . . | 181 |
| B.1.4 | Full LOTOS . . . . . | 184 |

## AUSTRALIAN/NEW ZEALAND STANDARD

**Information technology — Enhancements to LOTOS (E-LOTOS)****1 Scope**

This International Standard defines the syntax and semantics of the enhanced LOTOS language (ISO 8807), named E-LOTOS. E-LOTOS is used for the formal description of the behavioural aspects of distributed and concurrent systems in general and in the area of open distributed processing in particular.

**2 Conformance**

A formal specification written in E-LOTOS conforms to the requirements of this International Standard if and only if it is derivable according to the syntactic rules defined in clause 5 and the semantics is unambiguously derivable from the semantic definition in clause 7.

**3 Normative reference**

The following normative document contains provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent edition of the normative document indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO 8807, *Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour*

**4 Terms, definitions and notation**

This clause describes the concrete syntax for E-LOTOS. Here we use a notation similar to Extended Backus-Naur format which is summarized in the following table:

| <i>Symbol</i>             | <i>meaning</i>                                                  |
|---------------------------|-----------------------------------------------------------------|
|                           | alternative definition                                          |
| <i>D, K, V, S, RT ...</i> | terminal and non-terminal symbols                               |
| <b>text</b>               | E-LOTOS keywords                                                |
| ::=                       | definition                                                      |
| *                         | repetition of the preceding syntactic unit (zero or more times) |
| ()                        | grouping syntactic units                                        |
| []                        | optional (may or may not occur)                                 |

It is worth noting that the metalinguistic '()' and '[]' must not be confused with the E-LOTOS symbols '[', ']', '(', ')', and '|'. In the syntax, these E-LOTOS symbols appear between single quotes. Other symbols (: , -> , etc.) will appear without quotes to keep the International Standard readable.

Syntax for predefined types are described in clause 10.

## 5 E-LOTOS grammar

### 5.1 Lexical Structure

This clause describes the lexical units (tokens) of E-LOTOS.

#### 5.1.1 Character set

Characters are divided into several classes denoted by the nonterminals below:

`<character> ::= <letter> | <digit> | <special-character> | <blank-character>`

`<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"  
| "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"  
| "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"`

`<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"`

`<normal-character> ::= <letter> | <digit>`

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

`<blank-character> ::= SP | HT | VT | FF | LF | NL | CR`

`<character> ::= <letter> | <digit> | <special-character> | <blank-character>`

Formally, no distinction is made between different versions of the same character, such as capitalized, bold, italic, etc.

### 5.1.2 Comments and separators

Let `<any-string-of-text>` represent any string of characters not containing the substring `"*)"`. Then, comments are defined as:

```
<comment> ::= "(" <any-string-of-text> ")"
```

Comments are no part of E-LOTOS description. Comments may be inserted anywhere between two other lexical units or left out, except when they play the role of separators. Basically, a comment may be substituted by a SP.

A separator is defined as:

```
<separator> ::= <blank-character> | <comment>
```

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

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

### 5.1.3 Identifiers

All E-LOTOS objects are designated by *identifiers*. An identifier begins with a letter, possibly followed by any number of digits, letters and underscore `"_"` symbol, and ends with a digit or a letter.

```
<identifier> ::= <letter> ( [ "_" ] <normal-character> )*
```

All characters in an identifier are significant, and there is no limit to their length.

*Special identifiers* are defined in order to allow an intuitive notation for mathematical operators, such as `"+"`, `"/"`, etc. They are built from special characters, normal characters and digits.

```
<special-identifier> ::=
    <special-character> <special-character>*
    | <digit> ( [ "_" ] <normal-character> )*
```

### 5.1.4 Reserved words

The reserved words for the language are:

|             |           |            |            |               |
|-------------|-----------|------------|------------|---------------|
| and         | andalso   | any        | as         | behavior      |
| behaviour   | block     | break      | by         | case          |
| choice      | conc      | dis        | do         | else          |
| elsif       | endcase   | endch      | endconc    | enddis        |
| endeqns     | endexit   | endexn     | endfor     | endfunc       |
| endfullsync | endgen    | endhide    | endif      | endint        |
| endinter    | endloop   | endmod     | endpar     | endproc       |
| endren      | endsel    | endspec    | endsuspend | endtrap       |
| endtype     | endval    | endvar     | endwhile   | eqns          |
| etc         | exception | exceptions | exit       | external      |
| for         | forall    | fullsync   | function   | gate          |
| gates       | generic   | hide       | i          | if            |
| imports     | in        | infix      | inter      | interface     |
| is          | loop      | module     | none       | null          |
| ofsort      | opns      | orelse     | out        | par           |
| process     | procs     | raise      | raises     | rename        |
| renames     | renaming  | sel        | signal     | specification |
| stop        | suspend   | then       | trap       | type          |
| types       | value     | values     | var        | wait          |
| while       |           |            |            |               |

The reserved lexical tokens for the language are:

```
( ) { } , .
; : | ? ! =>
= <> := [ ] ->
# >
```

### 5.1.5 Identifiers classes

The terminals of the language syntax are:

| <i>identifier domain</i> | <i>meaning</i>           | <i>abbreviation</i> |
|--------------------------|--------------------------|---------------------|
| Con                      | constructor identifier   | <i>C</i>            |
| Const                    | constant identifier      | <i>K</i>            |
| Exc                      | exception identifier     | <i>X</i>            |
| Fun                      | function identifier      | <i>F</i>            |
| Gat                      | gate identifier          | <i>G</i>            |
| GenId                    | generics identifiers     | <i>gen-id</i>       |
| IntId                    | interface identifiers    | <i>int-id</i>       |
| ModId                    | module identifiers       | <i>mod-id</i>       |
| Proc                     | process identifier       | $\Pi$               |
| Spec                     | specification identifier | $\Sigma$            |
| Typ                      | type identifier          | <i>S</i>            |
| Var                      | variable identifier      | <i>V</i>            |

### 5.1.6 Non-terminals classes

The non-terminals for the language are:

| <i>symbol domain</i> | <i>meaning</i>                | <i>abbreviation</i> |
|----------------------|-------------------------------|---------------------|
| APL                  | actual parameter list         | <i>APL</i>          |
| Behav                | behaviour expression          | <i>B</i>            |
| BehavTerm            | behaviour term                | <i>BT</i>           |
| BehavAtom            | behaviour atom                | <i>BA</i>           |
| BMatch               | behaviour match               | <i>BM</i>           |
| Decl                 | declaration                   | <i>D</i>            |
| EMatch               | expression match              | <i>EM</i>           |
| EqnDecList           | equations declaration list    | <i>eqn-dec-list</i> |
| EqnDec               | simple equations declaration  | <i>eqn-dec</i>      |
| Exp                  | expression                    | <i>E</i>            |
| ExpAtom              | expression atom               | <i>EA</i>           |
| FPL                  | formal parameter list         | <i>FPL</i>          |
| GPL                  | gate parameter list           | <i>GPL</i>          |
| InParList            | in parameter list             | <i>IPL</i>          |
| InPar                | in parameter                  | <i>IP</i>           |
| IntBody              | interface body                | <i>i-body</i>       |
| IntExp               | interface expressions         | <i>int-exp</i>      |
| LocVar               | local variables               | <i>LV</i>           |
| ModBody              | module body                   | <i>m-body</i>       |
| ModExp               | module expressions            | <i>mod-exp</i>      |
| ModPar               | module formal parameters      | <i>MP</i>           |
| Pat                  | pattern                       | <i>P</i>            |
| RecModExp            | record module expressions     | <i>RME</i>          |
| Reninst              | module renaming/instantiation | <i>reninst</i>      |
| RPat                 | record pattern                | <i>RP</i>           |
| RTyExp               | record type expression        | <i>RT</i>           |
| RVal                 | record value expression       | <i>RN</i>           |
| Spec                 | specification                 | <i>spec</i>         |
| TopDec               | top-level declarations        | <i>top-dec</i>      |
| TyExp                | type expression               | <i>T</i>            |
| Val                  | value expression              | <i>N</i>            |
| XPL                  | exception parameter list      | <i>XPL</i>          |

## 5.2 Syntax of the language

### 5.2.1 Specification

$spec ::= [top-dec]$  *specification* (spec1)  
**specification**  $\Sigma$  [**imports**  $mod-exp(, mod-exp)^*$ ] **is**  
 $[gates\ G[:T](, G[:T])^* \quad [exceptions\ X[:T](, X[:T])^*]$   
 $(behaviour\ B) \mid (value\ E)$   
**endspec**

The default gate list is [], the default gate type is **any**, the default exception list is [] and the default exception type is ().

### 5.2.2 Top-level declaration

|                      |                                                                                                                                                                       |                                   |            |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|------------|
| $top\text{-}dec ::=$ | <b>module</b> $mod\text{-}id$ $[: int\text{-}exp]$<br>[ <b>imports</b> $mod\text{-}exp(, mod\text{-}exp)^*$ ] <b>is</b> $m\text{-}body$ <b>endmod</b>                 | <i>module</i>                     | (top-dec1) |
|                      | <b>module</b> $mod\text{-}id$ $: int\text{-}exp$ <b>is external</b> <b>endmod</b>                                                                                     | <i>external module</i>            | (top-dec2) |
|                      | <b>generic</b> $gen\text{-}id$ $'( ' MP ' )'$ $[: int\text{-}exp]$<br>[ <b>imports</b> $mod\text{-}exp(, mod\text{-}exp)^*$ ] <b>is</b> $m\text{-}body$ <b>endgen</b> | <i>generic module</i>             | (top-dec3) |
|                      | <b>interface</b> $int\text{-}id$<br>[ <b>imports</b> $int\text{-}exp(, int\text{-}exp)^*$ ] <b>is</b> $int\text{-}exp$ <b>endint</b>                                  | <i>interface</i>                  | (top-dec4) |
|                      | $top\text{-}dec$ $top\text{-}dec$                                                                                                                                     | <i>sequential top declaration</i> | (top-dec5) |

### 5.2.3 Module body

|                     |                  |                          |           |
|---------------------|------------------|--------------------------|-----------|
| $m\text{-}body ::=$ | $D$              | <i>block declaration</i> | (m-body1) |
|                     | $mod\text{-}exp$ | <i>module expression</i> | (m-body2) |

### 5.2.4 Module expression

|                      |                                                                                                     |                               |            |
|----------------------|-----------------------------------------------------------------------------------------------------|-------------------------------|------------|
| $mod\text{-}exp ::=$ | $mod\text{-}id$ $[: int\text{-}exp]$ [ <b>renaming</b> $'( ' reninst ' )'$ ]                        | <i>module aliasing</i>        | (mod-exp1) |
|                      | $gen\text{-}id$ $'( ' [ RME ] ' )'$ $[: int\text{-}exp]$<br>[ <b>renaming</b> $'( ' reninst ' )'$ ] | <i>actualization</i>          | (mod-exp2) |
|                      | $gen\text{-}id$ $'( ' reninst ' )'$                                                                 | <i>renaming/instantiation</i> | (mod-exp3) |

### 5.2.5 Module formal parameters

|          |                                    |                       |       |
|----------|------------------------------------|-----------------------|-------|
| $MP ::=$ | $mod\text{-}id$ $: int\text{-}exp$ | <i>single</i>         | (MP1) |
|          | $MP$ , $MP$                        | <i>disjoint union</i> | (MP2) |

### 5.2.6 Interface expressions

|                      |                                                                    |                        |            |
|----------------------|--------------------------------------------------------------------|------------------------|------------|
| $int\text{-}exp ::=$ | $int\text{-}id$                                                    | <i>interface id</i>    | (int-exp1) |
|                      | $'[ ' int\text{-}id$ <b>renaming</b> $'( ' reninst ' )'$ $']'$     | <i>simple renaming</i> | (int-exp2) |
|                      | $'[ ' i\text{-}body$ [ <b>renaming</b> $'( ' reninst ' )'$ ] $']'$ | <i>explicit body</i>   | (int-exp3) |

### 5.2.7 Record module expression

|           |                                            |                       |        |
|-----------|--------------------------------------------|-----------------------|--------|
| $RME ::=$ | $mod\text{-}id \Rightarrow mod\text{-}exp$ | <i>single</i>         | (RME1) |
|           | $RME$ , $RME$                              | <i>disjoint union</i> | (RME2) |

### 5.2.8 Interface body

|                     |                                                                                                                                                                                                                                    |                                      |            |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|------------|
| $i\text{-body} ::=$ | <b>type</b> $S$                                                                                                                                                                                                                    | <i>abstract data type</i>            | (i-body1)  |
|                     | <b>type</b> $S$ <b>renames</b> $T$ <b>endtype</b>                                                                                                                                                                                  | <i>type synonym</i>                  | (i-body2)  |
|                     | <b>type</b> $S$ <b>is</b> $C[ \text{' ( ' } RT \text{' ) ' } ] ( \text{'   ' } C[ \text{' ( ' } RT \text{' ) ' } ] )^*$ <b>endtype</b>                                                                                             | <i>constructed type</i>              | (i-body3)  |
|                     | <b>type</b> $S$ <b>is</b> $\text{' ( ' } RT \text{' ) ' }$ <b>endtype</b>                                                                                                                                                          | <i>named record type declaration</i> | (i-body4)  |
|                     | <b>process</b> $\Pi$ $[ \text{' [ ' } G[: T](, G[: T])^* \text{' ] ' } ]$<br>$[ \text{' ( ' } FPL \text{' ) ' } ]$<br>$[ \textbf{raises} \text{' [ ' } X[: T](, X[: T])^* \text{' ] ' } ]$                                         | <i>process</i>                       | (i-body5)  |
|                     | <b>function</b> $F$ $[ \text{' ( ' } FPL \text{' ) ' } ]$<br>$[ : T ]$<br>$[ \textbf{raises} \text{' [ ' } X[: T](, X[: T])^* \text{' ] ' } ]$                                                                                     | <i>function</i>                      | (i-body6)  |
|                     | <b>function</b> $F$ <b>infix</b> $\text{' ( ' } [( \textbf{in}   \textbf{out} )] V : T, [( \textbf{in}   \textbf{out} )] V : T \text{' ) ' }$<br>$[ : T ]$<br>$[ \textbf{raises} \text{' [ ' } X[: T](, X[: T])^* \text{' ] ' } ]$ | <i>infix function</i>                | (i-body7)  |
|                     | <b>value</b> $V : S$                                                                                                                                                                                                               | <i>constant value</i>                | (i-body8)  |
|                     | <b>eqns</b> $eqn\text{-}dec\text{-}list$ <b>endeqns</b>                                                                                                                                                                            | <i>equations</i>                     | (i-body9)  |
|                     | $i\text{-body}$ $i\text{-body}$                                                                                                                                                                                                    | <i>sequential</i>                    | (i-body10) |

### 5.2.9 Formal parameter list

|           |                    |                         |        |
|-----------|--------------------|-------------------------|--------|
| $FPL ::=$ | $V : T$            | <i>singleton</i>        | (fpl1) |
|           | <b>in</b> $V : T$  | <i>input singleton</i>  | (fpl2) |
|           | <b>out</b> $V : T$ | <i>output singleton</i> | (fpl3) |
|           | $FPL, FPL$         | <i>disjoint union</i>   | (fpl4) |

### 5.2.10 Renaming/Instantiation

|               |                                                      |                                     |            |
|---------------|------------------------------------------------------|-------------------------------------|------------|
| $reninst ::=$ | <b>types</b> $S := S(, S := S)^*$                    | <i>types</i>                        | (reninst1) |
|               | <b>opns</b> $(C := C   F := F)(, C := C   F := F)^*$ | <i>constructors and functions</i>   | (reninst2) |
|               | <b>procs</b> $\Pi := \Pi(, \Pi := \Pi)^*$            | <i>processes</i>                    | (reninst3) |
|               | <b>values</b> $V := V(, V := V)^*$                   | <i>value</i>                        | (reninst4) |
|               | $reninst, reninst$                                   | <i>renaming/instantiation tuple</i> | (reninst5) |

### 5.2.11 Equations declaration

|                                  |                                                              |                              |                 |
|----------------------------------|--------------------------------------------------------------|------------------------------|-----------------|
| $eqn\text{-}dec\text{-}list ::=$ | <b>[forall</b> $RT$ ]<br>$eqn\text{-}dec (eqn\text{-}dec)^*$ | <i>equations declaration</i> | (eqn-dec-list1) |
|                                  | $eqn\text{-}dec\text{-}list$ $eqn\text{-}dec\text{-}list$    | <i>sequential</i>            | (eqn-dec-list2) |

### 5.2.12 Simple equations declaration

$eqn-dec ::= \text{ofsort } T [\text{forall } RT ] E$  simple equations declaration (eqn-dec1)

### 5.2.13 Declarations

$D ::=$

- type**  $S$  **renames**  $T$  **endtype** type synonym (D1)
- | **type**  $S$  **is**  $C[(' RT ')] (' | ' C[(' RT ')] )^* \text{endtype}$  type declaration (D2)
- | **type**  $S$  **is**  $(' RT ')$  **endtype** named records type declaration (D3)
- | **process**  $\Pi$   $[(' G[: T](, G[: T])^* ')]$  process with in/out parameters (D4)  
 $[(' FPL ')]$   
 $[\text{raises } (' X[: T](, X[: T])^* ')]$
- is**  $B$
- endproc**
- | **function**  $F$   $[(' FPL ')] [: T]$  function with in/out parameters (D5)  
 $[\text{raises } (' X[: T](, X[: T])^* ')]$   
 $\text{is } E$
- endfunc**
- | **function**  $F$  **infix**  $(' [(in|out)]V: T, [(in|out)]V: T ')$  infix function (D6)  
 $[: T]$   
 $[\text{raises } (' X[: T](, X[: T])^* ')]$
- is**  $E$
- endfunc**
- | **value**  $V : S$  **is**  $E$  **endval** constant value declaration (D7)
- |  $D D$  sequential declaration (D8)

### 5.2.14 Expressions

$E ::=$

- $EA$  expression atom (E1)
- |  $EA ; E$  sequential composition (E2)

### 5.2.15 Expression Atoms

$EA ::=$

- $P := EA$  assignment (EA1)
- | **trap** trap (EA2)  
 $(\text{exception } X [(' IPL ')] \text{ is } E \text{ endexn})^*$   
 $[\text{exit } [P] \text{ is } E \text{ endexit}]$   
 $\text{in } E$   
**endtrap**
- | **var**  $V: T[: = E](, V: T[: = E])^* \text{ in}$  variable declaration (EA3)  
 $E$   
**endvar**

|                                                               |                                            |
|---------------------------------------------------------------|--------------------------------------------|
| <b>rename</b>                                                 | <i>renaming</i> (EA4)                      |
| (signal $X$ [ $'( ' IPL ' )'$ ] is $X$ [ $E$ ]) <sup>*</sup>  |                                            |
| <b>in</b> $E$                                                 |                                            |
| <b>endren</b>                                                 |                                            |
| <b>loop</b> $E$ <b>endloop</b>                                | <i>iteration</i> (EA5)                     |
| <b>loop</b> $X$ [ $:T$ ] <b>in</b>                            | <i>breakable iteration</i> (EA6)           |
| $E$                                                           |                                            |
| <b>endloop</b>                                                |                                            |
| <b>while</b> $E$ <b>do</b>                                    | <i>while iteration</i> (EA7)               |
| $E$                                                           |                                            |
| <b>endwhile</b>                                               |                                            |
| <b>for</b> $E$ <b>while</b> $E$ <b>by</b> $E$ <b>do</b>       | <i>for iteration</i> (EA8)                 |
| $E$                                                           |                                            |
| <b>endfor</b>                                                 |                                            |
| <b>case</b> $E$ <b>is</b> $EM$ <b>endcase</b>                 | <i>case</i> (EA9)                          |
| <b>case</b> $'( ' E(,E)^* ' )'$ <b>is</b> $EM$ <b>endcase</b> | <i>case with tuples</i> (EA10)             |
| <b>if</b> $E$ <b>then</b> $E$                                 | <i>if-then-else</i> (EA11)                 |
| ( <b>elsif</b> $E$ <b>then</b> $E$ ) <sup>*</sup>             |                                            |
| [ <b>else</b> $E$ ]                                           |                                            |
| <b>endif</b>                                                  |                                            |
| $F$ [ $'( ' APL ' )'$ ] [ $'[ ' XPL ' ]'$ ]                   | <i>function instantiation</i> (EA12)       |
| $(EA P)$ $F$ $(EA P)$ [ $'[ ' XPL ' ]'$ ]                     | <i>infix function instantiation</i> (EA13) |
| <b>raise</b> $X$ [ $'( ' E ' )'$ ]                            | <i>raising exception</i> (EA14)            |
| <b>break</b> [ $X$ ] [ $'( ' E ' )'$ ]                        | <i>breaking iteration</i> (EA15)           |
| $N$                                                           | <i>value</i> (EA16)                        |
| $EA$ <b>andalso</b> $EA$                                      | <i>conjunction</i> (EA17)                  |
| $EA$ <b>orelse</b> $EA$                                       | <i>disjunction</i> (EA18)                  |
| $EA = EA$                                                     | <i>equality</i> (EA19)                     |
| $EA <> EA$                                                    | <i>inequality</i> (EA20)                   |
| $EA . V$                                                      | <i>select field</i> (EA21)                 |
| $EA : T$                                                      | <i>explicit typing</i> (EA22)              |
| $'( ' E ' )'$                                                 | <i>parenthesized expression</i> (EA23)     |

### 5.2.16 Behaviour expressions

|            |                             |
|------------|-----------------------------|
| $B ::= BT$ | <i>behaviour term</i> (B1)  |
| $DisB$     | <i>disabling</i> (B2)       |
| $SyncB$    | <i>synchronization</i> (B3) |
| $ConcB$    | <i>concurrency</i> (B4)     |
| $SelB$     | <i>choice</i> (B5)          |
| $SuspendB$ | <i>suspend/resume</i> (B6)  |
| $InterB$   | <i>interleaving</i> (B7)    |

### 5.2.17 Disabling behaviour expression

|                                 |                       |         |
|---------------------------------|-----------------------|---------|
| $DisB ::= BT \ [ > \ BT$        | <i>singleton</i>      | (DisB1) |
| $\quad   \quad BT \ [ > \ DisB$ | <i>disjoint union</i> | (DisB2) |

### 5.2.18 Synchronization behaviour expression

|                                 |                        |          |
|---------------------------------|------------------------|----------|
| $SyncB ::= BT \    \ BT$        | <i>synchronization</i> | (SyncB1) |
| $\quad   \quad BT \    \ SyncB$ | <i>synchronization</i> | (SyncB2) |

### 5.2.19 Concurrency behaviour expression

|                                                |                    |          |
|------------------------------------------------|--------------------|----------|
| $ConcB ::= BT \   \ [G(,G)^*] \   \ BT$        | <i>concurrency</i> | (ConcB1) |
| $\quad   \quad BT \   \ [G(,G)^*] \   \ ConcB$ | <i>concurrency</i> | (ConcB2) |

### 5.2.20 Selection behaviour expression

|                                |               |         |
|--------------------------------|---------------|---------|
| $SelB ::= BT \ [] \ BT$        | <i>choice</i> | (SelB1) |
| $\quad   \quad BT \ [] \ SelB$ | <i>choice</i> | (SelB2) |

### 5.2.21 Suspend/Resume behaviour expression

|                                      |                       |             |
|--------------------------------------|-----------------------|-------------|
| $SuspendB ::= BT \ [X > \ BT$        | <i>suspend/resume</i> | (SuspendB1) |
| $\quad   \quad BT \ [X > \ SuspendB$ | <i>suspend/resume</i> | (SuspendB2) |

### 5.2.22 Interleaving behaviour expression

|                                   |                     |           |
|-----------------------------------|---------------------|-----------|
| $InterB ::= BT \     \ BT$        | <i>interleaving</i> | (InterB1) |
| $\quad   \quad BT \     \ InterB$ | <i>interleaving</i> | (InterB2) |

### 5.2.23 Behaviour term

|                             |                               |       |
|-----------------------------|-------------------------------|-------|
| $BT ::= BA$                 | <i>behaviour atom</i>         | (BT1) |
| $\quad   \quad BA \ ; \ BT$ | <i>sequential composition</i> | (BT2) |

### 5.2.24 Behaviour atom

|               |                                                                                                                                                                                                                    |                                             |        |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------|--------|
| <b>BA</b> ::= | <b>G</b> [ <b>P</b> ] [ <b>@P</b> ] [ ' [ <b>E</b> ' ] ' ]                                                                                                                                                         | <i>action</i>                               | (BA1)  |
|               | <b>i</b>                                                                                                                                                                                                           | <i>internal action</i>                      | (BA2)  |
|               | <b>null</b>                                                                                                                                                                                                        | <i>successful termination</i>               | (BA3)  |
|               | <b>stop</b>                                                                                                                                                                                                        | <i>inaction</i>                             | (BA4)  |
|               | <b>block</b>                                                                                                                                                                                                       | <i>time block</i>                           | (BA5)  |
|               | <b>wait</b> ' ( ' <b>E</b> ' ) ' ,                                                                                                                                                                                 | <i>delay</i>                                | (BA6)  |
|               | <b>P</b> := <b>EA</b>                                                                                                                                                                                              | <i>assignment</i>                           | (BA7)  |
|               | <b>P</b> := <b>any</b> <b>T</b> [ ' [ <b>E</b> ' ] ' ]                                                                                                                                                             | <i>nondeterministic assignment</i>          | (BA8)  |
|               | <b>dis</b> <b>DisB</b> <b>enddis</b>                                                                                                                                                                               | <i>bracketed disabled expression</i>        | (BA9)  |
|               | <b>fullsync</b> <b>SyncB</b> <b>endfullsync</b>                                                                                                                                                                    | <i>bracketed synchronization expression</i> | (BA10) |
|               | <b>conc</b> <b>ConcB</b> <b>endconc</b>                                                                                                                                                                            | <i>bracketed concurrency expression</i>     | (BA11) |
|               | <b>sel</b> <b>SelB</b> <b>endsel</b>                                                                                                                                                                               | <i>bracketed choice expression</i>          | (BA12) |
|               | <b>suspend</b> <b>SuspendB</b> <b>endsuspend</b>                                                                                                                                                                   | <i>bracketed suspend/resume expression</i>  | (BA13) |
|               | <b>inter</b> <b>InterB</b> <b>endinter</b>                                                                                                                                                                         | <i>bracketed interleaving expression</i>    | (BA14) |
|               | <b>choice</b> <b>P</b> [ ] <b>B</b> <b>endch</b>                                                                                                                                                                   | <i>choice over values</i>                   | (BA15) |
|               | <b>trap</b><br>( <b>exception</b> <b>X</b> [ ' ( ' <b>IPL</b> ' ) ' ] <b>is B</b> <b>endexn</b> ) *<br>[ <b>exit</b> [ <b>P</b> ] <b>is B</b> <b>endexit</b> ]<br><b>in B</b><br><b>endtrap</b>                    | <i>trap</i>                                 | (BA16) |
|               | <b>par</b> [ <b>G</b> # <b>n</b> ( , <b>G</b> # <b>n</b> ) * ] <b>in</b><br>' [ ' [ <b>G</b> ( , <b>G</b> ) * ] ' ] ' -> <b>B</b> (     ' [ ' [ <b>G</b> ( , <b>G</b> ) * ] ' ] ' -> <b>B</b> ) *<br><b>endpar</b> | <i>general parallel</i>                     | (BA17) |
|               | <b>par</b> <b>P</b> <b>in</b> <b>N</b>       <b>B</b> <b>endpar</b>                                                                                                                                                | <i>parallel over values</i>                 | (BA18) |
|               | <b>var</b> <b>V</b> : <b>T</b> [ : = <b>E</b> ] ( , <b>V</b> : <b>T</b> [ : = <b>E</b> ] ) * <b>in</b><br><b>B</b><br><b>endvar</b>                                                                                | <i>variable declaration</i>                 | (BA19) |
|               | <b>hide</b> <b>G</b> [ : <b>T</b> ] ( , <b>G</b> [ : <b>T</b> ] ) * <b>in</b><br><b>B</b><br><b>endhide</b>                                                                                                        | <i>gate hiding</i>                          | (BA20) |
|               | <b>rename</b><br>( ( <b>gate</b> <b>G</b> [ ( <b>IPL</b> ) ] <b>is G</b> [ <b>P</b> ] )   ( <b>signal</b> <b>X</b> [ ( <b>IPL</b> ) ] <b>is X</b> [ <b>E</b> ] ) *<br><b>in B</b><br><b>endren</b>                 | <i>renaming</i>                             | (BA21) |
|               | $\Pi$ ' [ ' [ <b>GPL</b> ] ' ] ' ,<br>' ( ' [ <b>APL</b> ] ' ) ' ,<br>[ ' [ ' <b>XPL</b> ' ] ' ]                                                                                                                   | <i>process instantiation</i>                | (BA22) |
|               | <b>loop</b> <b>B</b> <b>endloop</b>                                                                                                                                                                                | <i>iteration</i>                            | (BA23) |
|               | <b>loop</b> <b>X</b> [ : <b>T</b> ] <b>in</b><br><b>B</b><br><b>endloop</b>                                                                                                                                        | <i>breakable iteration</i>                  | (BA24) |
|               | <b>while</b> <b>E</b> <b>do</b><br><b>B</b><br><b>endwhile</b>                                                                                                                                                     | <i>while iteration</i>                      | (BA25) |

|                                                         |                                       |
|---------------------------------------------------------|---------------------------------------|
| <b>for</b> $E$ <b>while</b> $E$ <b>by</b> $E$ <b>do</b> | <i>for iteration</i> (BA26)           |
| $B$                                                     |                                       |
| <b>endfor</b>                                           |                                       |
| <b>case</b> $E$ <b>is</b>                               | <i>case</i> (BA27)                    |
| $BM$                                                    |                                       |
| <b>endcase</b>                                          |                                       |
| <b>case</b> $'( ' E ( , E ) ^* ' )'$ <b>is</b>          | <i>case with tuples</i> (BA28)        |
| $BM$                                                    |                                       |
| <b>endcase</b>                                          |                                       |
| <b>if</b> $E$ <b>then</b> $B$                           | <i>if-then-else</i> (BA29)            |
| ( <b>elsif</b> $E$ <b>then</b> $B$ )*                   |                                       |
| [ <b>else</b> $B$ ]                                     |                                       |
| <b>endif</b>                                            |                                       |
| <b>signal</b> $X$ [ $'( ' E ' )'$ ]                     | <i>signalling</i> (BA30)              |
| <b>raise</b> $X$ [ $'( ' E ' )'$ ]                      | <i>raising exception</i> (BA31)       |
| <b>break</b> [ $X$ ] [ $'( ' E ' )'$ ]                  | <i>breaking iteration</i> (BA32)      |
| $'( ' B ' )'$                                           | <i>parenthesized behaviour</i> (BA33) |

### 5.2.25 Type expressions

|             |                             |
|-------------|-----------------------------|
| $T ::= S$   | <i>type identifier</i> (T1) |
| <b>none</b> | <i>empty type</i> (T2)      |
| <b>any</b>  | <i>universal type</i> (T3)  |

### 5.2.26 Record type expressions

|                                                                    |                               |
|--------------------------------------------------------------------|-------------------------------|
| $RT ::= V \Rightarrow T ( , V \Rightarrow T ) ^* [ , \text{etc} ]$ | <i>named</i> (RT1)            |
| <b>etc</b>                                                         | <i>universal record</i> (RT2) |
| $T ( , T ) ^*$                                                     | <i>positional</i> (RT3)       |

### 5.2.27 Value expressions

|                |                                     |
|----------------|-------------------------------------|
| $N ::= K$      | <i>primitive constant</i> (N1)      |
| $V$            | <i>variables</i> (N2)               |
| $'( ' RN ' )'$ | <i>record values</i> (N3)           |
| $C [N]$        | <i>constructor application</i> (N4) |

### 5.2.28 Record value expressions

$$\begin{array}{ll}
 RN ::= V \Rightarrow N(, V \Rightarrow N)^* & \text{named (RN1)} \\
 | N, N(, N)^* & \text{positional (RN2)}
 \end{array}$$

### 5.2.29 Patterns

$$\begin{array}{ll}
 P ::= ' ( ' RP ' ) ' & \text{records (P1)} \\
 | \text{any} : T & \text{wildcard (P2)} \\
 | ?V & \text{variable (P3)} \\
 | !EA & \text{expression (P4)} \\
 | C [P] & \text{constructor application (P5)} \\
 | P : T & \text{explicit typing (P6)}
 \end{array}$$

### 5.2.30 Gate parameter list

$$\begin{array}{ll}
 GPL ::= G \Rightarrow G (, G \Rightarrow G)^* [ , \dots ] & \text{explicit instantiation (GPL1)} \\
 | \dots & \text{explicit instantiation (GPL2)} \\
 | G(, G)^* & \text{positional instantiation (GPL3)}
 \end{array}$$

Note that these rules require that “...” occurs at most once in a terminal position.

### 5.2.31 Actual parameter list

$$\begin{array}{ll}
 APL ::= V \Rightarrow (E|P) (, V \Rightarrow (E|P))^* [ , \dots ] & \text{explicit instantiation (APL1)} \\
 | \dots & \text{explicit instantiation (APL2)} \\
 | (E|P)(, (E|P))^* & \text{positional instantiation (APL3)}
 \end{array}$$

Note that these rules require that “...” occurs at most once in a terminal position.

### 5.2.32 Exception parameter list

$$\begin{array}{ll}
 XPL ::= X \Rightarrow X (, X \Rightarrow X)^* [ , \dots ] & \text{explicit instantiation (XPL1)} \\
 | \dots & \text{explicit instantiation (XPL2)} \\
 | X(, X)^* & \text{positional instantiation (XPL3)}
 \end{array}$$

Note that these rules require that “...” occurs at most once in a terminal position.

### 5.2.33 Record patterns

|                                                               |                         |
|---------------------------------------------------------------|-------------------------|
| $RP ::= V \Rightarrow P (, V \Rightarrow P)^* [, \text{etc}]$ | <i>named</i> (RP1)      |
| <b>etc</b>                                                    | <i>wildcard</i> (RP2)   |
| $P (, P)^*$                                                   | <i>positional</i> (RP3) |

### 5.2.34 Behaviour pattern matching

|                                            |                             |
|--------------------------------------------|-----------------------------|
| $BM ::= P [ ' [ ' E ' ] ' ] \rightarrow B$ | <i>single match</i> (BM1)   |
| $BM '   ' BM$                              | <i>multiple match</i> (BM2) |

### 5.2.35 Expression pattern matching

|                                            |                             |
|--------------------------------------------|-----------------------------|
| $EM ::= P [ ' [ ' E ' ] ' ] \rightarrow E$ | <i>single match</i> (EM1)   |
| $EM '   ' EM$                              | <i>multiple match</i> (EM2) |

### 5.2.36 In parameter list

|                       |                              |
|-----------------------|------------------------------|
| $IPL ::= IP (, IP)^*$ | <i>disjoint union</i> (IPL1) |
|-----------------------|------------------------------|

### 5.2.37 In parameter

|                              |                                       |
|------------------------------|---------------------------------------|
| $IP ::= V \Rightarrow [P:]T$ | <i>singleton</i> (IP1)                |
| <b>etc</b>                   | <i>wildcard</i> (IP2)                 |
| $P \text{ as } IP$           | <i>record match</i> (IP3)             |
| $[P:]T$                      | <i>tuple</i> (IP4)                    |
| $' ( ' IPL ' ) '$            | <i>parenthesized parameters</i> (IP5) |

with the restriction that **etc** can occur at most once.

## 6 E-LOTOS abstract syntax

### 6.1 Overview

In this clause, the translation from concrete syntax into abstract syntax is defined. This includes the syntactic sugar rules definition. Note that although abstract syntax is represented by text, the semantics will deal with an abstract syntax tree. In particular, brackets are used to represent such tree textually.

### 6.1.1 Syntactic sugar

In the grammars, non-primitive constructs (which are defined in terms of syntactic sugar for primitives) are marked with a ' $\star$ '. The terms added to the syntax are marked with a '+'. These grammars omit any **end**-keywords from the concrete grammar. For clarity, the whole rules are included.

Many of the constructs in the base language are defined as syntactic sugar, for example **if**-statements are defined as syntactic sugar for **case**-statement.

In addition, we define the following non-terminals as syntactic sugar:

| <i>symbol domain</i> | <i>meaning</i>           | <i>abbreviation</i> | <i>sugar for</i>          |
|----------------------|--------------------------|---------------------|---------------------------|
| APL                  | parameter list           | <i>APL</i>          | $E_1 P_1, \dots, E_n P_n$ |
| Exp                  | expression               | <i>E</i>            | <i>B</i>                  |
| ExpAtom              | expression atom          | <i>EA</i>           | <i>B</i>                  |
| EMatch               | expression match         | <i>EM</i>           | <i>BM</i>                 |
| FPL                  | formal parameter list    | <i>FPL</i>          | $V_1:T_1, \dots, V_n:T_n$ |
| GPL                  | gate parameter list      | <i>GPL</i>          | $G_1, \dots, G_n$         |
| XPL                  | exception parameter list | <i>XPL</i>          | $X_1, \dots, X_n$         |

**Syntax sugar context** Syntax sugar for process instantiation needs information from the proper process declaration. This information is used for:

- transforming explicit instantiation of gates, exceptions and values into positional instantiation
- solving the abbreviated parameters lists ("..." as -partial- instantiation list).
- change **out** parameters to local variable definition with value capture (via **trap**).

In other words, process instantiation in abstract syntax has only positional input parameters.

The context keeps position information of out parameter taken from process declarations and is used in process instantiations, and the formal gate, formal parameter, and exception lists. It is composed by judgements as  $S \vdash \Pi \Rightarrow Pos$ .

|                                                          |                             |
|----------------------------------------------------------|-----------------------------|
| $S ::= \Pi \Rightarrow Pos$                              | positional information (S1) |
| $\Pi \Rightarrow \mathbf{gates}([G_1, \dots, G_n])$      | formal gate list (S2)       |
| $\Pi \Rightarrow \mathbf{params}([V_1, \dots, V_p])$     | formal parameter list (S3)  |
| $\Pi \Rightarrow \mathbf{exceptions}([X_1, \dots, X_m])$ | formal exception list (S4)  |
| $S, S$                                                   | disjoint union (S5)         |

Intuitively, *Pos* is a set of indexes, and it stores the positions of **out** parameters.

### 6.1.2 Abstract syntax

The following constructions are defined:

- **after**(*N*): where *N* is a number.
- **start**(*N*): where *N* is a number.
- **exit**(*RN*): successful behaviour end with returning value.

Abstract syntax is a superset of the syntax defined by the grammar in Clause 5. However, to keep definitions shorter, it usually omits the closing reserved word (**endproc**, **endvar**, ...).

Some syntactic categories are extended in abstract syntax. So, we allow record types *RT*, record value expressions *RN*, and record patterns *RP* to be empty. See Subclauses 6.2.10, 6.2.11, and 6.2.15 respectively.

*RT* is considered a *T* in abstract syntax, see Subclause 6.2.9, although it is not allowed to express anonymous record types in E-LOTOS text, this rule ease the semantics definition.

## 6.2 Concrete to abstract syntactic translation

In this clause, we revisit only those categories that need translation. Each subclause is divided in an “Overview” subclause that includes the syntactic category, with marks for clauses that changes from concrete to abstract syntax (‘★’ for clauses which have a translation and sometimes are removed, ‘+’ for added clauses). Then, the translation for each clause is given. Besides, we include those category that modifies the syntax sugar context, need for translating **out** parameters.

Some categories are translated in several steps, as they may be defined as a translation into concrete syntax which will need further translation.

### 6.2.1 Interface body

#### 6.2.1.1 Overview

##### Concrete syntax

|               |                                                                                                                                                                                   |                                       |            |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------|------------|
| <i>i-body</i> | <b> ::= type S</b>                                                                                                                                                                | <i>abstract data type</i>             | (i-body1)  |
|               | <b>type S renames T endtype</b>                                                                                                                                                   | <i>type synonym</i>                   | (i-body2)  |
|               | <b>type S is C[ ’ ( ’ RT ’ ) ’ ] ( ’   ’ C[ ’ ( ’ RT ’ ) ’ ] ) * endtype</b>                                                                                                      | <i>constructed type</i>               | (i-body3)  |
|               | <b>type S is ’ ( ’ RT ’ ) ’ endtype</b>                                                                                                                                           | <i>named records type declaration</i> | (i-body4)  |
| ★             | <b>process Π</b> [ ’ [ ’ G[ : T ] ( , G[ : T ] ) * ] ’ ]<br>[ ’ ( ’ FPL ’ ) ’ ]<br>[ <b>raises</b> ’ [ ’ X[ : T ] ( , X[ : T ] ) * ] ’ ]                                          | <i>process</i>                        | (i-body5)  |
| ★             | <b>function F</b> [ ’ ( ’ FPL ’ ) ’ ]<br>[ : T ]<br>[ <b>raises</b> ’ [ ’ X[ : T ] ( , X[ : T ] ) * ] ’ ]                                                                         | <i>function</i>                       | (i-body6)  |
| ★             | <b>function F infix</b> ’ ( ’ [( <b>in</b>   <b>out</b> ) ] V : T , [( <b>in</b>   <b>out</b> ) ] V : T ’ ) ’<br>[ : T ]<br>[ <b>raises</b> ’ [ ’ X[ : T ] ( , X[ : T ] ) * ] ’ ] | <i>infix function</i>                 | (i-body7)  |
| ★             | <b>value V : S</b>                                                                                                                                                                | <i>constant value</i>                 | (i-body8)  |
|               | <b>eqns eqn-dec-list endeqns</b>                                                                                                                                                  | <i>equations</i>                      | (i-body9)  |
|               | <i>i-body i-body</i>                                                                                                                                                              | <i>sequential</i>                     | (i-body10) |
| +             | <b>process Π</b> [ ’ [ ’ G[ : T ] ( , G[ : T ] ) * ] ’ ]<br>[ ’ ( ’ V : T ( , V : T ) * ] ’ ) ’<br>: T<br>[ <b>raises</b> ’ [ ’ X[ : T ] ( , X[ : T ] ) * ] ’ ]                   | <i>process with return value</i>      | (i-body11) |

**function** declarations are synonymous with the equivalent **process** declaration. **value** declarations are synonymous with the equivalent **function** declarations.

The last category is added to join functions and processes in the abstract syntax in just one category, a process with a return value and just in parameters.

#### 6.2.1.2 Process declaration

##### Concrete syntax

**process Π** [ ’ [ ’ G<sub>1</sub>[ : T<sub>1</sub> ] , … , G<sub>n</sub>[ : T<sub>n</sub> ] ’ ]  
[ ’ ( ’ FPL ’ ) ’ ]  
[ **raises** [ X<sub>1</sub>[ : T<sub>1</sub> ] , … , X<sub>m</sub>[ : T<sub>m</sub> ] ] ]

The default gate list is [], the default gate type is **any**, the default parameter list is (), the default exception list is [] and the default exception type is ().

## Context

$$\overline{\mathcal{S} \vdash \Pi \Rightarrow \{j \mid \text{out } V_j : T_j \in FPL\}}$$

$$\overline{\mathcal{S} \vdash \Pi \Rightarrow \text{gates}([G_1, \dots, G_n])}$$

$$\overline{\mathcal{S} \vdash \Pi \Rightarrow \text{params}([V_1, \dots, V_p])}$$

$$\overline{\mathcal{S} \vdash \Pi \Rightarrow \text{exceptions}([X_1, \dots, X_m])}$$

with  $FPL = [\text{in}|\text{out}] V_1 : T'_1, \dots, [\text{in}|\text{out}] V_p : T'_p$

## Syntax sugar

$$\left( \begin{array}{l} \text{process } \Pi \quad [G_1 : T_1, \dots, G_n : T_n] \\ \quad (FPL) \\ \text{raises } [X_1 : T''_1, \dots, X_m : T''_m] \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{process } \Pi \quad [G_1 : T_1, \dots, G_n : T_n] \\ \quad (I) \\ \quad : OT \\ \text{raises } [X_1 : T''_1, \dots, X_m : T''_m] \end{array} \right)$$

where (ordered with ascendant indexes):

$I = \{V_i : T_i \mid [\text{in}] V_i : T_i \in FPL\}$  and

$OT = (\{T_i \mid \text{out } V_i : T_i \in FPL\})$ . If there is no **out** parameters, the return type is **none**.

### 6.2.1.3 Function declaration

#### Concrete syntax

```
function F [ ' ( ' FPL ' ) ' ]
           [ : T ]
           [ raises ' [ ' X[: T] ( , X[: T] ) * ' ] ' ]
```

The default parameter list is (), the default result type is (), the default exception list is [] and the default exception type is ().

#### Syntax sugar

$$\left( \begin{array}{l} \text{function } F (FPL) \\ \text{raises } [X_1 : T_1, \dots, X_n : T_n] \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{process } F (FPL) \\ \text{raises } [X_1 : T_1, \dots, X_n : T_n] \end{array} \right)$$

$$\left( \begin{array}{l} \text{function } F (FPL) : T \\ \text{raises } [X_1 : T_1, \dots, X_n : T_n] \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{process } F (I) : T \\ \text{raises } [X_1 : T_1, \dots, X_n : T_n] \end{array} \right)$$

The last definition is valid when  $FPL$  contains only in parameters, in other case the function declaration does not type check. Here  $I$  is obtained from  $FPL$  removing the **in** keywords.

### 6.2.1.4 Infix function declaration

#### Concrete syntax

```
function F infix ' ( ' [(in|out)]V:T , [(in|out)]V:T ' ) '
                [ : T ]
                [ raises ' [ ' X[: T] ( , X[: T] ) * ' ] ' ]
```

The default result type is (), the default exception list is [] and the default exception type is ().

### Syntax sugar

$$\left( \begin{array}{l} \text{function } F \text{ infix } (FP_1, FP_2) : T \\ \text{raises } [X_1 : T_1, \dots, X_n : T_n] \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{process } F (FP_1, FP_2) : T \\ \text{raises } [X_1 : T_1, \dots, X_n : T_n] \end{array} \right)$$

where  $FP_i = [\text{in}|\text{out}]V_i : T_i$ .

#### 6.2.1.5 Constant value

##### Concrete syntax

value  $V : S$

##### Syntax sugar

$$(\text{value } V : S) \stackrel{\text{def}}{=} (\text{function } V() : S)$$

### 6.2.2 Formal parameter list

#### 6.2.2.1 Overview

##### Concrete syntax

|                                |                                |
|--------------------------------|--------------------------------|
| $FPL ::= V : T$                | <i>singleton</i> (FPL1)        |
| $\star \mid \text{in } V : T$  | <i>input singleton</i> (FPL2)  |
| $\star \mid \text{out } V : T$ | <i>output singleton</i> (FPL3) |
| $\mid FPL, FPL$                | <i>disjoint union</i> (FPL4)   |

This category disappears in the abstract syntax, see Subclause 6.2.1.2, and it is rewritten as a list of **in** typed parameters. **out** parameters are dealt with as local variable declaration with value capturing via **trap**, see Subclauses 6.2.3.2 and 6.2.8.7.

### 6.2.3 Declarations

#### 6.2.3.1 Overview

##### Concrete syntax

|                                                                                                                                                                             |                                             |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------|
| $D ::= \text{type } S \text{ renames } T \text{ endtype}$                                                                                                                   | <i>type synonym</i> (D1)                    |
| $\mid \text{type } S \text{ is } C[(' RT ')] (' \mid ' C[(' RT ')] )^* \text{ endtype}$                                                                                     | <i>type declaration</i> (D2)                |
| $\mid \text{type } S \text{ is } ' (' RT ' ) ' \text{ endtype}$                                                                                                             | <i>named records type declaration</i> (D3)  |
| $\star \mid \text{process } \Pi \quad \begin{array}{l} [ ' [ ' [G : T] ( , G : T )^* ] ' ] \\ [ ' ( ' FPL ' ) ' ] \\ [\text{raises } [X : T] ( , X : T )^* ] \end{array}$   | <i>process with in/out parameters</i> (D4)  |
| $\quad \text{is } B$                                                                                                                                                        |                                             |
| $\quad \text{endproc}$                                                                                                                                                      |                                             |
| $\star \mid \text{function } F \quad \begin{array}{l} [ ' ( ' FPL ' ) ' ] [ : T ] \\ [\text{raises } ' [ ' X : T ] ( , X : T )^* ' ] ' ] \\ \text{is } E \\ \text{endfunc}$ | <i>function with in/out parameters</i> (D5) |

|   |                                                                                                                                                                                                 |                                        |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| ★ | <b>function</b> $F$ <b>infix</b> ' ( ' [(in out)] $V:T$ , [(in out)] $V:T$ ' ) '<br>[ : $T$ ]<br>[raises ' [ ' $X:T$ ] ( , $X:T$ ) * ' ] ' ]                                                    | <i>infix function</i> (D6)             |
|   | is $E$<br><b>endfunc</b>                                                                                                                                                                        |                                        |
| ★ | <b>value</b> $V : S$ <b>is</b> $E$ <b>endval</b>                                                                                                                                                | <i>constant value declaration</i> (D7) |
|   | $D D$                                                                                                                                                                                           | <i>sequential declaration</i> (D8)     |
| + | <b>process</b> $\Pi$ [ ' [ ' [ $G[:T]$ ( , $G[:T]$ ) * ] ' ] ' ]<br>[ ' ( ' $V:T$ ( , $V:T$ ) * ' ) ' ]<br>: $T$<br>[raises ' [ ' $X:T$ ] ( , $X:T$ ) * ' ] ' ]<br><br>is $B$<br><b>endproc</b> | <i>process with return value</i> (D9)  |

**Syntax sugar** The **function** declarations are synonymous with the equivalent **process** declaration. Also, **value** constants are syntax sugar of functions with arguments ().

### 6.2.3.2 Process declaration

#### Concrete syntax

```

process  $\Pi$    [ ' [ '  $G_1[:T_1]$  , ... ,  $G_n[:T_n]$  ' ] ' ]
               [ ' ( '  $FPL$  ' ) ' ]
               [raises [  $X_1[:T_1'']$  , ... ,  $X_m[:T_m'']$  ] ]
                 

               is  $B$ 
               endproc

```

The default gate list is [], the default gate type is **any**, the default parameter list is (), the default exception list is [] and the default exception type is ().

#### Context

$$\overline{\mathcal{S} \vdash \Pi \Rightarrow \{j \mid \text{out } V_j : T_j \in FPL\}}$$

$$\overline{\mathcal{S} \vdash \Pi \Rightarrow \text{gates}([G_1, \dots, G_n])}$$

$$\overline{\mathcal{S} \vdash \Pi \Rightarrow \text{params}([V_1, \dots, V_p])}$$

$$\overline{\mathcal{S} \vdash \Pi \Rightarrow \text{exceptions}([X_1, \dots, X_m])}$$

where  $FPL = [\text{in|out } V_1 : T_1', \dots, \text{in|out } V_p : T_p']$

### Syntax sugar

$$\left( \begin{array}{l} \text{process } \Pi \quad [G_1 : T_1, \dots, G_n : T_n] \\ \quad (FPL) \\ \text{raises } [X_1 : T_1'', \dots, X_m : T_m''] \\ \\ \text{is } B \\ \text{endproc} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{process } \Pi \quad [G_1 : T_1, \dots, G_n : T_n] \\ \quad (I) \\ \quad : OT \\ \quad \text{raises } [X_1 : T_1'', \dots, X_m : T_m''] \\ \\ \text{is} \\ \quad \text{var } O \text{ in} \\ \quad \quad B ; \text{exit}(RV) \\ \quad \text{endvar} \\ \text{endproc} \end{array} \right)$$

where (every tuple is ordered with ascendant indexes):

$I = (V_{i_1} : T_{i_1}, \dots, V_{i_k} : T_{i_k})$  with  $\{V_{i_1} : T_{i_1}, \dots, V_{i_k} : T_{i_k}\} = \{V_i : T_i \mid [\text{in}] V_i : T_i \in FPL\}$ ,

$O = \{V_i : T_i \mid \text{out } V_i : T_i \in FPL\}$ ,

$OT = '(\{T_i \mid \text{out } V_i : T_i \in FPL\})'$  (if there is no **out** parameters, the return type is **none**), and

$RV = \{V_i \mid \text{out } V_i : T_i \in FPL\}$ .

### 6.2.3.3 Function declaration

#### Concrete syntax

```
function F [ ' ( ' FPL ' ) ' ] [ : T ]
  [ raises ' [ ' X [ : T ] ( , X [ : T ] ) * ' ' ] ' ]
  is E
endfunc
```

The default parameter list is  $()$ , the default exception list is  $[]$  and the default exception type is  $()$ .

#### Syntax sugar

$$\left( \begin{array}{l} \text{function } F \text{ ( } FPL \text{ )} \\ \quad \text{raises } [X_1 : T_1, \dots, X_n : T_n] \\ \quad \text{is } E \\ \text{endfunc} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{process } F \text{ ( } FPL \text{ )} \\ \quad \text{raises } [X_1 : T_1, \dots, X_n : T_n] \\ \quad \text{is } E \\ \text{endproc} \end{array} \right)$$

$$\left( \begin{array}{l} \text{function } F \text{ ( } FPL \text{ ) : } T \\ \quad \text{raises } [X_1 : T_1, \dots, X_n : T_n] \\ \quad \text{is } E \\ \text{endfunc} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{process } F \text{ ( } I \text{ ) : } T \\ \quad \text{raises } [X_1 : T_1, \dots, X_n : T_n] \\ \quad \text{is } E \\ \text{endproc} \end{array} \right)$$

The last definition is valid when  $FPL$  contains only in parameters, in other case the function declaration does not type check. Here  $I$  is obtained from  $FPL$  removing the **in** keywords.

### 6.2.3.4 Infix function declaration

#### Concrete syntax

```
function F infix ' ( ' [in|out]V1 : T1 , [in|out]V2 : T2 ' ) '
  [ : T ]
  [ raises ' [ ' X [ : T ] ( , X [ : T ] ) * ' ' ] ' ]
  is E
endfunc
```

The default exception list is  $[]$  and the default exception type is  $()$ .

**Syntax sugar**

$$\left( \begin{array}{l} \text{function } F \text{ infix } (FP_1, FP_2) : T \\ \quad \text{raises } [X_1 : T_1, \dots, X_n : T_n] \\ \quad \text{is } E \\ \text{endfunc} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{process } F (FP_1, FP_2) : T \\ \quad \text{raises } [X_1 : T_1, \dots, X_n : T_n] \\ \quad \text{is } E \\ \text{endproc} \end{array} \right)$$

where  $FP_i = [\text{in}|\text{out}]V_i : T_i$ .

**6.2.3.5 Constant value declarations****Concrete syntax**

**value**  $V : S$  **is**  $E$  **endval**

**Syntax sugar**

$$(\text{value } V : S \text{ is } E \text{ endval}) \stackrel{\text{def}}{=} (\text{function } V() : S \text{ is } E \text{ endfunc})$$

**6.2.4 Expressions****6.2.4.1 Overview****Concrete syntax**

$E ::= EA$   
 $\quad | EA ; E$

*expression atom* (E1)

*sequential composition* (E2)

**Syntax sugar** See notes on syntax sugar for clause 6.2.5.

**6.2.5 Expression atoms****6.2.5.1 Overview****Concrete syntax**

$EA ::= P := EA$   
 $\quad | \text{trap}$   
 $\quad \quad (\text{exception } X [\text{'(' IPL ')'}] \text{ is } E \text{ endexn})^*$   
 $\quad \quad [\text{exit } [P] \text{ is } E \text{ endexit}]$   
 $\quad \quad \text{in } E$   
 $\quad \quad \text{endtrap}$   
 $\quad | \text{var } V : T [:=E] (, V : T [:=E])^* \text{ in}$   
 $\quad \quad E$   
 $\quad \quad \text{endvar}$   
 $\quad | \text{rename}$   
 $\quad \quad (\text{signal } X [\text{'(' IPL ')'}] \text{ is } X [E])^*$   
 $\quad \quad \text{in } E$   
 $\quad \quad \text{endren}$   
 $\quad \star | \text{loop } E \text{ endloop}$

*assignment* (EA1)

*trap* (EA2)

*variable declaration* (EA3)

*renaming* (EA4)

*iteration* (EA5)

|                                                                                                                   |                                            |
|-------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| ★   <b>loop</b> $X[:T]$ <b>in</b><br>$E$<br><b>endloop</b>                                                        | <i>breakable iteration</i> (EA6)           |
| ★   <b>while</b> $E$ <b>do</b><br>$E$<br><b>endwhile</b>                                                          | <i>while iteration</i> (EA7)               |
| ★   <b>for</b> $E$ <b>while</b> $E$ <b>by</b> $E$ <b>do</b><br>$E$<br><b>endfor</b>                               | <i>for iteration</i> (EA8)                 |
| ★   <b>case</b> $E$ <b>is</b> $EM$ <b>endcase</b>                                                                 | <i>case</i> (EA9)                          |
| ★   <b>case</b> $'(E(,E)^*)'$ <b>is</b> $EM$ <b>endcase</b>                                                       | <i>case with tuples</i> (EA10)             |
| ★   <b>if</b> $E$ <b>then</b> $E$<br>( <b>elsif</b> $E$ <b>then</b> $E$ )*<br>[ <b>else</b> $E$ ]<br><b>endif</b> | <i>if-then-else</i> (EA11)                 |
| ★   $F[( '([APL]) ' ) ] [ '([XPL]) ' ]$                                                                           | <i>function instantiation</i> (EA12)       |
| ★   $(EA P) F (EA P) [ '([XPL]) ' ]$                                                                              | <i>infix function instantiation</i> (EA13) |
| ★   <b>raise</b> $X[( 'E ' )]$                                                                                    | <i>raising exception</i> (EA14)            |
| ★   <b>break</b> $[X] [ 'E ' ]$                                                                                   | <i>breaking iteration</i> (EA15)           |
| ★   $N$                                                                                                           | <i>value</i> (EA16)                        |
| ★   $EA$ <b>andalso</b> $EA$                                                                                      | <i>conjunction</i> (EA17)                  |
| ★   $EA$ <b>orelse</b> $EA$                                                                                       | <i>disjunction</i> (EA18)                  |
| ★   $EA = EA$                                                                                                     | <i>equality</i> (EA19)                     |
| ★   $EA <> EA$                                                                                                    | <i>inequality</i> (EA20)                   |
| ★   $EA . V$                                                                                                      | <i>select field</i> (EA21)                 |
| ★   $EA : T$                                                                                                      | <i>explicit typing</i> (EA22)              |
| $'(E)'$                                                                                                           | <i>parenthesized expression</i> (EA23)     |

**Syntax sugar** Some of these clauses are syntax sugar, some of them with an explicit translation as described in following subclauses and some with the same translation that applies for behaviours. The rest are particular cases of behaviours only capable of performing termination ( $\delta$ ) or exception ( $X$ ) transitions, and not internal ( $i$ ), gate ( $G$ ) or delay ( $\epsilon$ ) transitions.

We translate each expression of type  $T$  into a behaviour of type **exit**( $T$ ). Expressions are deterministic.

Most of the translations are straightforward, since they are the same as the behaviour parts. We only give the non-trivial translations here. Note that infix function instantiations are firstly translated into prefix function instantiations, then considered as behaviour.

### 6.2.5.2 Infix function instantiation

#### Concrete syntax

$(EA|P) F (EA|P) [ '([XPL]) ' ]$

The default exception list is  $[]$ .

**Syntax sugar**

$$(AP_1 \ F \ AP_2 \ [XPL]) \stackrel{\text{def}}{=} (F \ (AP_1, AP_2) \ [XPL])$$

where  $AP_i = EA_i | P_i$ .

**6.2.5.3 Value****Concrete syntax**

$N$

**Syntax sugar**

$$(N) \stackrel{\text{def}}{=} (\text{exit}(\$1 \Rightarrow N))$$

**6.2.5.4 Conjunction****Concrete syntax**

$EA \text{ andalso } EA$

**Syntax sugar**

$$(EA_1 \text{ andalso } EA_2) \stackrel{\text{def}}{=} (\text{if } EA_1 \text{ then } EA_2 \text{ else false})$$

**6.2.5.5 Disjunction****Concrete syntax**

$EA \text{ orelse } EA$

**Syntax sugar**

$$(EA_1 \text{ orelse } EA_2) \stackrel{\text{def}}{=} (\text{if } EA_1 \text{ then true else } EA_2)$$

**6.2.5.6 Equality****Concrete syntax**

$EA = EA$

**Syntax sugar**

$$(EA_1 = EA_2) \stackrel{\text{def}}{=} (\text{case}(EA_1, EA_2) \text{ is } (?x, ?y) \rightarrow \text{case } x \text{ is } !y \rightarrow \text{true} \mid \text{any} \rightarrow \text{false})$$

**6.2.5.7 Inequality****Concrete syntax**

$EA <> EA$

**Syntax sugar**

$$(EA_1 <> EA_2) \stackrel{\text{def}}{=} (\text{if } EA_1 = EA_2 \text{ then false else true})$$

#### 6.2.5.8 Field select

##### Concrete syntax

$$EA . V$$

##### Syntax sugar

$$(EA . V) \stackrel{\text{def}}{=} (\text{case } EA \text{ is } (V \Rightarrow ?x, \text{etc}) \rightarrow x)$$

#### 6.2.5.9 Explicit typing

##### Concrete syntax

$$EA : T$$

##### Syntax sugar

$$(EA : T) \stackrel{\text{def}}{=} (\text{case } EA:T \text{ is } ?x \rightarrow x)$$

### 6.2.6 Behaviour expressions

#### Concrete Syntax

|            |                             |
|------------|-----------------------------|
| $B ::= BT$ | <i>behaviour term</i> (B1)  |
| $DisB$     | <i>disabling</i> (B2)       |
| $SyncB$    | <i>synchronization</i> (B3) |
| $ConcB$    | <i>concurrency</i> (B4)     |
| $SelB$     | <i>choice</i> (B5)          |
| $SuspendB$ | <i>suspend/resume</i> (B6)  |
| $InterB$   | <i>interleaving</i> (B7)    |

**Syntax sugar** Note that this category is introduced to prioritize operators, and it is just the union of other categories. Those categories that need syntactic translation are included in the next sections. The rest will be revisited in the semantic Clause 9.

### 6.2.7 Interleaving behaviour expression

#### Concrete syntax

|                              |                               |
|------------------------------|-------------------------------|
| $InterB \star ::= BT     BT$ | <i>interleaving</i> (InterB1) |
| $\star   BT     InterB$      | <i>interleaving</i> (InterB2) |

#### Syntax sugar

$$(B_1 ||| B_2) \stackrel{\text{def}}{=} (B_1 | [] | B_2)$$

## 6.2.8 Behaviour atom

### 6.2.8.1 Overview

#### Concrete syntax

|          |                                                                                                                         |                                             |        |
|----------|-------------------------------------------------------------------------------------------------------------------------|---------------------------------------------|--------|
| $BA ::=$ | $\star G [P] [\textcircled{P}] [ \text{'[ ' E ' ]' } ]$                                                                 | <i>action</i>                               | (BA1)  |
| +        | $  G P \textcircled{P} E \text{ start}(N)$                                                                              | <i>action</i>                               | (BA2)  |
|          | $  \mathbf{i}$                                                                                                          | <i>internal action</i>                      | (BA3)  |
|          | $  \mathbf{null}$                                                                                                       | <i>successful termination</i>               | (BA4)  |
|          | $  \mathbf{stop}$                                                                                                       | <i>inaction</i>                             | (BA5)  |
|          | $  \mathbf{block}$                                                                                                      | <i>time block</i>                           | (BA6)  |
|          | $  \mathbf{wait} \text{'( ' E ' )' }$                                                                                   | <i>delay</i>                                | (BA7)  |
|          | $  P := EA$                                                                                                             | <i>assignment</i>                           | (BA8)  |
|          | $  P := \mathbf{any} T [ \text{'[ ' E ' ]' } ]$                                                                         | <i>nondeterministic assignment</i>          | (BA9)  |
| $\star$  | $  \mathbf{dis} DisB \mathbf{enddis}$                                                                                   | <i>bracketed disabled expression</i>        | (BA10) |
| $\star$  | $  \mathbf{fullsync} SyncB \mathbf{endfullsync}$                                                                        | <i>bracketed synchronisation expression</i> | (BA11) |
| $\star$  | $  \mathbf{conc} ConcB \mathbf{endconc}$                                                                                | <i>bracketed concurrency expression</i>     | (BA12) |
| $\star$  | $  \mathbf{sel} SelB \mathbf{endsel}$                                                                                   | <i>bracketed choice expression</i>          | (BA13) |
| $\star$  | $  \mathbf{suspend} SuspendB \mathbf{endsuspend}$                                                                       | <i>bracketed suspend/resume expression</i>  | (BA14) |
| $\star$  | $  \mathbf{inter} InterB \mathbf{endinter}$                                                                             | <i>bracketed interleaving expression</i>    | (BA15) |
| $\star$  | $  \mathbf{choice} P [ ] B \mathbf{endch}$                                                                              | <i>choice over values</i>                   | (BA16) |
| +        | $  \mathbf{choice} P \mathbf{after}(N) [ ] B$                                                                           | <i>choice over values</i>                   | (BA17) |
|          | $  \mathbf{trap}$                                                                                                       | <i>trap</i>                                 | (BA18) |
|          | $  \quad (\mathbf{exception} X [ \text{'( ' IPL ' )' } ] \mathbf{is} B \mathbf{endexn})^*$                              |                                             |        |
|          | $  \quad [\mathbf{exit} [P] \mathbf{is} B \mathbf{endexit}]$                                                            |                                             |        |
|          | $  \quad \mathbf{in} B$                                                                                                 |                                             |        |
|          | $  \quad \mathbf{endtrap}$                                                                                              |                                             |        |
|          | $  \quad \mathbf{par} [G\#n(, G\#n)^*] \mathbf{in}$                                                                     | <i>general parallel</i>                     | (BA19) |
|          | $  \quad \text{'[ ' } [G(, G)^*] \text{' ]' } \rightarrow B (    \text{'[ ' } [G(, G)^*] \text{' ]' } \rightarrow B)^*$ |                                             |        |
|          | $  \quad \mathbf{endpar}$                                                                                               |                                             |        |
|          | $  \quad \mathbf{par} P \mathbf{in} N     B \mathbf{endpar}$                                                            | <i>parallel over values</i>                 | (BA20) |
| $\star$  | $  \quad \mathbf{var} V : T [ := E ] (, V : T [ := E ])^* \mathbf{in}$                                                  | <i>variable declaration</i>                 | (BA21) |
|          | $  \quad B$                                                                                                             |                                             |        |
|          | $  \quad \mathbf{endvar}$                                                                                               |                                             |        |
|          | $  \quad \mathbf{hide} G [ : T ] (, G [ : T ])^* \mathbf{in}$                                                           | <i>gate hiding</i>                          | (BA22) |
|          | $  \quad B$                                                                                                             |                                             |        |
|          | $  \quad \mathbf{endhide}$                                                                                              |                                             |        |
|          | $  \quad \mathbf{rename}$                                                                                               | <i>renaming</i>                             | (BA23) |
|          | $  \quad ( (\mathbf{gate} G [ (IPL) ] \mathbf{is} G [P]) \mid (\mathbf{signal} X [ (IPL) ] \mathbf{is} X [E]) )^*$      |                                             |        |
|          | $  \quad \mathbf{in} B$                                                                                                 |                                             |        |
|          | $  \quad \mathbf{endren}$                                                                                               |                                             |        |
| $\star$  | $  \quad \Pi \text{'[ ' } [GPL] \text{' ]' }$                                                                           | <i>process instantiation</i>                | (BA24) |
|          | $  \quad \text{'( ' } [APL] \text{' )' }$                                                                               |                                             |        |
|          | $  \quad [ \text{'[ ' } XPL \text{' ]' } ]$                                                                             |                                             |        |

|                                                                                                          |                                                               |
|----------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|
| +   $\Pi$ ' [ ' [G(,G)*] ' ] ' ,<br>' ( ' [E(,E)*] ' ) ' ,<br>[ ' [ ' X(X)* ' ] ' ]                      | <i>process instantiation with positional arguments</i> (BA25) |
| *   <b>loop</b> B <b>endloop</b>                                                                         | <i>iteration</i> (BA26)                                       |
| *   <b>loop</b> X [:T] <b>in</b><br>B<br><b>endloop</b>                                                  | <i>breakable iteration</i> (BA27)                             |
| *   <b>while</b> E <b>do</b><br>B<br><b>endwhile</b>                                                     | <i>while iteration</i> (BA28)                                 |
| *   <b>for</b> E <b>while</b> E <b>by</b> E <b>do</b><br>B<br><b>endfor</b>                              | <i>for iteration</i> (BA29)                                   |
| <b>case</b> E <b>is</b><br>BM<br><b>endcase</b>                                                          | <i>case</i> (BA30)                                            |
| +   <b>case</b> E :T <b>is</b><br>BM<br><b>endcase</b>                                                   | <i>case</i> (BA31)                                            |
| <b>case</b> ' ( ' E(,E)* ' ) ' <b>is</b><br>BM<br><b>endcase</b>                                         | <i>case with tuples</i> (BA32)                                |
| *   <b>if</b> E <b>then</b> B<br>( <b>elsif</b> E <b>then</b> B ) *<br>[ <b>else</b> B ]<br><b>endif</b> | <i>if-then-else</i> (BA33)                                    |
| <b>signal</b> X [ ' ( ' E ' ) ' ]                                                                        | <i>signalling</i> (BA34)                                      |
| *   <b>raise</b> X [ ' ( ' E ' ) ' ]                                                                     | <i>raising exception</i> (BA35)                               |
| *   <b>break</b> [X] [ ' ( ' E ' ) ' ]                                                                   | <i>breaking iteration</i> (BA36)                              |
| *   ' ( ' B ' ) ' ,                                                                                      | <i>parenthesized behaviour</i> (BA37)                         |
| +   <b>exit</b> [(RN)]                                                                                   | <i>Successful termination with return value</i> (BA38)        |

Note that some clause are just syntax sugar, with straightforward translation.

### 6.2.8.2 Action

#### Concrete syntax

$G [P] [ @P ] [ ' [ ' E ' ] ' ]$

Default values are ( ), @any:time, and [true] respectively.

#### Syntax sugar

$(G P_1 @P_2 [E]) \stackrel{\text{def}}{=} (G (P_1) @P_2 E \text{ start}(0))$

### 6.2.8.3 Bracketed disabled expression

#### Concrete syntax

**dis** DisB **endis**

**Syntax sugar**

$$(\mathbf{dis\ DisB\ enddis}) \stackrel{\text{def}}{=} ((\mathbf{DisB}))$$

All the bracketed behaviour expressions (**conc...endconc**, **sel...endsel**) will be translated in the same way.

**6.2.8.4 Choice over values****Concrete syntax**

**choice**  $P \ [] \ B$  **endch**

**Syntax sugar**

$$(\mathbf{choice\ } P \ [] \ B \ \mathbf{endch}) \stackrel{\text{def}}{=} (\mathbf{choice\ } P \ \mathbf{after}(0) \ [] \ B)$$

**6.2.8.5 Variable declaration****Concrete syntax**

**var**  $V_1 : T_1 [: = E_1], \dots, V_n : T_n [: = E_n]$  **in**  $B$  **endvar**

**Syntax sugar**

$$(\mathbf{var\ } \vec{DV} \ \mathbf{in\ } B \ \mathbf{endvar}) \stackrel{\text{def}}{=} (\mathbf{var\ } \vec{DV}' \ \mathbf{in\ } B'; B \ \mathbf{endvar})$$

where for each  $DV_i$  in  $\vec{DV}$ , we have:

$DV_i = V_i : T_i$  or  $DV_i = V_i : T_i [: = E_i]$ ,

$DV'_i = V_i : T_i$  and

$B' = ?V_{j_1} : = E_{j_1}; \dots ; ?V_{j_m} : = E_{j_m}$  where

$\{j_1, \dots, j_m\} = \{k | V_k : T_k [: = E_k] \in \vec{DV}\}$  and  $j_1, \dots, j_m$  are in ascending order.

**6.2.8.6 Process instantiation with abbreviated parameter lists****Concrete syntax**

$\Pi \text{ ' } [ \text{GPL} ] \text{ ' } \text{ ' } ( \text{ ' } [ \text{APL} ] \text{ ' } ) \text{ ' } [ \text{ ' } [ \text{XPL} ] \text{ ' } ]$

**Syntax sugar**

$\mathcal{S} \vdash \Pi \Rightarrow \mathbf{gates}([G_1, \dots, G_n])$

$\mathcal{S} \vdash \Pi \Rightarrow \mathbf{params}([V_1, \dots, V_p])$

$\mathcal{S} \vdash \Pi \Rightarrow \mathbf{exceptions}([X_1, \dots, X_m])$

$$(\Pi \ [ \text{GPL} ] \ ( \text{APL} ) \ [ \text{XPL} ]) \stackrel{\text{def}}{=} (\Pi \ [g_1, \dots, g_n] \ (ep_1, \dots, ep_p) \ [x_1, \dots, x_m])$$

where

$\text{GPL} = [G_{k1} \Rightarrow G'_{k1}, \dots, G_{kp} \Rightarrow G'_{kp}, \dots] \quad G_{ki} \in \{G_1, \dots, G_n\}$

$g_i = \begin{cases} G'_i & \text{iff } G_i \Rightarrow G'_i \in \text{GPL} \\ G_i & \text{otherwise} \end{cases}$

$\text{APL} = [V_{j1} \Rightarrow (E|P)'_{j1}, \dots, V_{js} \Rightarrow (E|P)'_{js}, \dots] \quad V_{ji} \in \{V_1, \dots, V_p\}$

$ep_i = \begin{cases} E'_i & \text{iff } V_i \Rightarrow E'_i \in \text{APL} \\ P'_i & \text{iff } V_i \Rightarrow P'_i \in \text{APL} \\ V_i & \text{otherwise} \end{cases}$

$\text{XPL} = [X_{l1} \Rightarrow X'_{l1}, \dots, X_{lq} \Rightarrow X'_{lq}, \dots] \quad X_{li} \in \{X_1, \dots, X_p\}$

$x_i = \begin{cases} X'_i & \text{iff } X_i \Rightarrow X'_i \in \text{XPL} \\ X_i & \text{otherwise} \end{cases}$

This syntactic sugar is applied only when there is explicit instantiation (via “=> ” and/or “...”). Explicit instantiation may exist only for gates, parameters or exception or any combination of them: the appropriate translation will be applied.

#### 6.2.8.7 Process instantiation with in/out parameters

##### Concrete syntax

$$\Pi \rightarrow [\rightarrow [G(,G)^*] \rightarrow] \rightarrow (\rightarrow [APL] \rightarrow) \rightarrow [\rightarrow [X(,X)^* \rightarrow] \rightarrow]$$

The default gate and exception lists are the empty list []. Before solving in/out parameters, dot notation for abbreviated parameters lists should have been solved (see Subclause 6.2.8.6).

##### Syntax sugar

$$\frac{S \vdash \Pi \Rightarrow Pos}{\left( \Pi [\vec{G}] (AP_1, \dots, AP_n) [\vec{X}] \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{trap exit ?x is } (RP) := x \\ \text{in } \Pi [\vec{G}] (RE) [\vec{X}] \end{array} \right)}$$

where

$$RP = (AP_{i_1}, \dots, AP_{i_k}) \text{ with } i_1 < i_2 < \dots < i_k \text{ and } \{i_1, \dots, i_k\} = Pos$$

$$RE = (AP_{j_1}, \dots, AP_{j_l}) \text{ with } j_1 < j_2 < \dots < j_l \text{ and } \{j_1, \dots, j_l\} = \{1, \dots, n\} - Pos$$

Note that this translation is only defined whenever (RP) is indeed an irrefutable Record Pattern and RE is indeed a Record Expression. Otherwise, the original behaviour is declared to be syntactically incorrect.

#### 6.2.8.8 Iteration

##### Concrete syntax

**loop B endloop**

There is a default exception, named **inner**.

**Syntax sugar** The final (semantic) *loop* is written in *italic* to avoid confusion.

$$\left( \begin{array}{l} \text{loop} \\ B \\ \text{endloop} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{trap} \\ \text{exception inner is exit} \\ \text{in} \\ \text{loop} \\ B \\ \text{endloop} \\ \text{endtrap} \end{array} \right)$$

#### 6.2.8.9 Breakable iteration

##### Concrete syntax

**loop X [: T] in B endloop**

The default exception type is ().

**Syntax sugar** The final (semantic) *loop* is written in *italic* to avoid confusion.

$$\left( \begin{array}{l} \text{loop } X \text{ in} \\ B \\ \text{endloop} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{trap} \\ \text{exception } X \text{ is exit} \\ \text{in} \\ \text{loop} \\ B \\ \text{endloop} \\ \text{endtrap} \end{array} \right)$$

$$\left( \begin{array}{l} \text{loop } X : T \text{ in} \\ B \\ \text{endloop} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{trap} \\ \text{exception } X (?x:T) \text{ is exit } (x) \\ \text{in} \\ \text{loop} \\ B \\ \text{endloop} \\ \text{endtrap} \end{array} \right)$$

#### 6.2.8.10 while iteration

**Concrete syntax**

**while** *E* **do** *B* **endwhile**

**Syntax sugar**

$$\left( \begin{array}{l} \text{while } E \text{ do} \\ B \\ \text{endwhile} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{loop} \\ \text{if } E \text{ then} \\ B \\ \text{else} \\ \text{break} \\ \text{endif} \\ \text{endloop} \end{array} \right)$$

#### 6.2.8.11 for iteration

**Concrete syntax**

**for** *E* **while** *E* **by** *E* **do** *B* **endfor**

**Syntax sugar**

$$\left( \begin{array}{l} \text{for } E_1 \text{ while } E_2 \text{ by } E_3 \text{ do} \\ B \\ \text{endfor} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} E_1; \\ \text{loop} \\ \text{if } E_2 \text{ then} \\ B; \\ E_3 \\ \text{else} \\ \text{break} \\ \text{endif} \\ \text{endloop} \end{array} \right)$$

### 6.2.8.12 Case

#### Concrete syntax

|                                |                                                             |
|--------------------------------|-------------------------------------------------------------|
| <b>case</b> <i>E</i> <b>is</b> | <b>case</b> ' ( ' <i>E</i> ( , <i>E</i> ) * ' ) ' <b>is</b> |
| <i>BM</i>                      | <i>BM</i>                                                   |
| <b>endcase</b>                 | <b>endcase</b>                                              |

To ensure that the match is exhaustive, a final **any**:any -> **raise** Match clause is added.

**Syntax sugar** The final (semantic) *case* is written in italic to avoid confusion with infinite recursion.

$$\left( \begin{array}{l} \text{case } E \text{ is} \\ \quad BM \\ \text{endcase} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{case } E \text{ is} \\ \quad BM \\ \quad | \text{ any } \rightarrow \text{raise Match} \\ \text{endcase} \end{array} \right)$$

$$\left( \begin{array}{l} \text{case } (\vec{E}) \text{ is} \\ \quad BM \\ \text{endcase} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{case } (\vec{E}) \text{ is} \\ \quad BM \\ \quad | \text{ any } \rightarrow \text{raise Match} \\ \text{endcase} \end{array} \right)$$

### 6.2.8.13 If-then-else

#### Concrete syntax

**if** *E* **then** *B* [**else** *B*] **endif**

The default **else** clause is **exit**.

#### Syntax sugar

$$(\text{if } E \text{ then } B_1 \text{ else } B_2 \text{ endif}) \stackrel{\text{def}}{=} (\text{case } E : \text{bool is true} \rightarrow B_1 \mid \text{false} \rightarrow B_2 \text{ endcase})$$

### 6.2.8.14 if-then-elsif-else

#### Concrete syntax

**if** *E* **then** *B* (**elsif** *E* **then** *B*) \* [**else** *B*] **endif**

The default **else** clause is **exit**.

#### Syntax sugar

$$\left( \begin{array}{l} \text{if } E_1 \text{ then } B_1 \\ \quad \text{elsif } E_2 \text{ then } B_2 \\ \quad \text{else } B_3 \\ \text{endif} \end{array} \right) \stackrel{\text{def}}{=} \left( \begin{array}{l} \text{if } E_1 \text{ then } B_1 \\ \quad \text{else} \\ \quad \quad \text{if } E_2 \text{ then } B_2 \\ \quad \quad \text{else } B_3 \\ \quad \text{endif} \\ \text{endif} \end{array} \right)$$

### 6.2.8.15 Raising exception

#### Concrete syntax

**raise** *X* [ ' ( ' *E* ' ) ' ]

The default expression is ().

**Syntax sugar**

$$(\text{raise } X(E)) \stackrel{\text{def}}{=} (\text{signal } X(E); \text{block})$$

**6.2.8.16 Breaking iteration****Concrete syntax**

**break** [*X*] [*'( ' E ' )'*]

The default exception name is **inner** and the default expression is ().

**Syntax sugar**

$$(\text{break } X) \stackrel{\text{def}}{=} (\text{raise } X)$$

$$(\text{break } X(E)) \stackrel{\text{def}}{=} (\text{raise } X(E))$$

$$(\text{break}) \stackrel{\text{def}}{=} (\text{raise inner})$$

$$(\text{break } (E)) \stackrel{\text{def}}{=} (\text{raise inner}(E))$$

**6.2.8.17 Successful termination**

A new behaviour term is added in the abstract syntax, just to allow process definition and other terms to return values. Note that this category belongs to the abstract syntax, so the user is not allowed to use it: it may appear just as a result of a syntactic translation or in the semantics.

**abstract syntax**

**exit** [(*RN*)]

The default termination value is ().

**6.2.9 Type expressions****Concrete syntax**

|                       |                             |
|-----------------------|-----------------------------|
| <i>T</i> ::= <i>S</i> | <i>type identifier</i> (T1) |
| <b>none</b>           | <i>empty type</i> (T2)      |
| <b>any</b>            | <i>universal type</i> (T3)  |

**Abstract syntax**

|                         |                             |
|-------------------------|-----------------------------|
| <i>T</i> ::= <i>S</i>   | <i>type identifier</i> (T1) |
| <b>none</b>             | <i>empty type</i> (T2)      |
| <b>any</b>              | <i>universal type</i> (T3)  |
| +   <i>'( ' RT ' )'</i> | <i>records</i> (T4)         |

## 6.2.10 Record type expressions

### 6.2.10.1 Overview

#### Concrete syntax

$$RT ::= V \Rightarrow T (, V \Rightarrow T)^* [, \text{etc}]$$

$$| \text{ etc}$$

$$* | T (, T)^*$$

*named* (RT1)  
*universal record* (RT2)  
*positional* (RT3)

#### Abstract syntax

$$RT ::= V \Rightarrow T (, V \Rightarrow T)^* [, \text{etc}]$$

$$| \text{ etc}$$

$$+ |$$

*named* (RT1)  
*universal record* (RT2)  
*empty record* (RT3)

### 6.2.10.2 Positional

#### Concrete syntax

$$T(, T)^*$$

#### Syntax sugar

$$(T_1, \dots, T_n) \stackrel{\text{def}}{=} (\$1 \Rightarrow T_1, \dots, \$n \Rightarrow T_n)$$

## 6.2.11 Record value expressions

### 6.2.11.1 Overview

#### Concrete syntax

$$RN ::= V \Rightarrow N (, V \Rightarrow N)^*$$

$$* | N, N(, N)^*$$

*named* (RN1)  
*positional* (RN2)

#### Abstract syntax

$$RN ::= V \Rightarrow N (, V \Rightarrow N)^*$$

$$+ |$$

*named* (RN1)  
*empty record* (RN2)

### 6.2.11.2 Positional

#### Concrete syntax

$$N(, N)^*$$

### Syntax sugar

$$(N_1, \dots, N_n) \stackrel{\text{def}}{=} (\$1 \Rightarrow N_1, \dots, \$n \Rightarrow N_n)$$

#### 6.2.12 Gate parameter list

$$\begin{array}{ll} \text{GPL } \star ::= & G \Rightarrow G(, G \Rightarrow G)^* [, \dots] & \text{explicit instantiation (GPL1)} \\ & \star | \dots & \text{implicit instantiation (GPL2)} \\ & | G(, G)^* & \text{positional instantiation (GPL3)} \end{array}$$

See Subclause 6.2.8.6, as syntax sugar for explicit instantiation needs context information (formal parameter list of the process to be instantiated).

#### 6.2.13 Actual parameter list

$$\begin{array}{ll} \text{APL } \star ::= & V \Rightarrow (E|P) (, V \Rightarrow (E|P))^* [, \dots] & \text{explicit instantiation (APL1)} \\ & \star | \dots & \text{implicit instantiation (APL2)} \\ & | (E|P)(, (E|P))^* & \text{positional instantiation (APL3)} \end{array}$$

See Subclause 6.2.8.6, as syntax sugar for explicit instantiation needs context information (formal parameter list of the process to be instantiated).

#### 6.2.14 Exception parameter list

$$\begin{array}{ll} \text{XPL } \star ::= & X \Rightarrow X(, X \Rightarrow X)^* [, \dots] & \text{explicit instantiation (XPL1)} \\ & \star | \dots & \text{implicit instantiation (XPL2)} \\ & | X(, X)^* & \text{positional instantiation (XPL3)} \end{array}$$

See Subclause 6.2.8.6, as syntax sugar for explicit instantiation needs context information (formal parameter list of the process to be instantiated).

#### 6.2.15 Record patterns

##### 6.2.15.1 Overview

##### Concrete syntax

$$\begin{array}{ll} \text{RP } ::= & V \Rightarrow P (, V \Rightarrow P)^* [, \text{etc}] & \text{named (RP1)} \\ & | \text{etc} & \text{wildcard (RP2)} \\ & \star | P(, P)^* & \text{positional (RP3)} \end{array}$$

**Abstract syntax**

|                                                                 |                           |
|-----------------------------------------------------------------|---------------------------|
| $RP ::= V \Rightarrow P (, V \Rightarrow P)^* [ , \text{etc} ]$ | <i>named</i> (RP1)        |
| $\quad \quad \quad   \text{ etc}$                               | <i>wildcard</i> (RP2)     |
| $\quad \quad \quad +   P \text{ as } RP$                        | <i>record match</i> (RP3) |
| $\quad \quad \quad +  $                                         | <i>empty record</i> (RP4) |

**6.2.15.2 Positional****Concrete syntax**

$$P(, P)^*$$
**Syntax sugar**

$$(P_1, \dots, P_n) \stackrel{\text{def}}{=} (\$1 \Rightarrow P_1, \dots, \$n \Rightarrow P_n)$$
**6.2.16 In parameter list****Concrete syntax**

$$IP (, IP)^*$$
**Syntax sugar**

$$(IP_1, \dots, IP_n) \stackrel{\text{def}}{=} (\$1 \Rightarrow IP_1, \dots, \$n \Rightarrow IP_n)$$
**6.2.17 In parameter****6.2.17.1 Overview****Concrete syntax**

|                                    |                                       |
|------------------------------------|---------------------------------------|
| $IP \star ::= V \Rightarrow [P:]T$ | <i>singleton</i> (IP1)                |
| $\quad \star   \text{ etc}$        | <i>wildcard</i> (IP2)                 |
| $\quad \star   P \text{ as } IP$   | <i>record match</i> (IP3)             |
| $\quad \star   [P:]T$              | <i>tuple</i> (IP4)                    |
| $\quad \star   ' ( ' IPL ' ) ' ,$  | <i>parenthesized parameters</i> (IP5) |

with the restriction that **etc** can occur at most once.

Each parameter list is translated to a typed record pattern of the form  $?\$argv \text{ as } RP : RT$

**6.2.17.2 Singleton parameter list****Concrete syntax**

$$V \Rightarrow [P:]T$$

The default pattern is **any**.

**Syntax sugar**

$$((V \Rightarrow P:T)) \stackrel{\text{def}}{=} (?\$argv \text{ as } (V \Rightarrow P) : (V \Rightarrow T))$$

### 6.2.17.3 Wildcard

#### Concrete syntax

**etc**

#### Syntax sugar

$$(\mathbf{etc}) \stackrel{\text{def}}{=} (? \$ \text{argv as etc} : \mathbf{etc})$$

### 6.2.17.4 Record match

#### Concrete syntax

$P \text{ as } IP$

#### Syntax sugar

$$(P \text{ as } ? \$ \text{argv as } RP : RT) \stackrel{\text{def}}{=} (? \$ \text{argv as } P \text{ as } RP : RT)$$

### 6.2.17.5 Tuple parameter list

#### Concrete syntax

$[P:]T$

The default pattern is **any**.

#### Syntax sugar

$$((P:T)) \stackrel{\text{def}}{=} ((\$1 \Rightarrow P:T))$$

### 6.2.17.6 Parenthesized parameters

#### Concrete syntax

$\text{' } ( \text{' IPL ' } ) \text{' }$

#### Syntax sugar

$$((IPL)) \stackrel{\text{def}}{=} ((\$1 \Rightarrow IPL))$$

## 7 E-LOTOS semantics

### 7.1 Overview

This clause describes the semantic objects and rules used for describing the E-LOTOS static and dynamic semantics.

At the semantics level, there exist some constructions which are not available in the grammar: they are defined just to ease the (static and dynamic) semantics definition. The following are defined:

- **after**: used to define the semantics of time evolution, see Subclause 9.2.16. Used as abstract syntax.
- **start**: local clock in semantics of actions, see Subclause 9.2.7. It records how long an action has been offered. Used as abstract syntax.
- **exit**: successful behaviour end with return value. Used as abstract syntax. Its semantics definition is in Subclause 9.2.10.

Besides, the following apply:

$$\mu ::= a \mid \delta \quad a ::= G \mid X \mid i$$

We assume the following types declared:

**type** bool **is** true | false  
**type** List **is** Nil | cons(**any**, List)

## 7.2 Static Semantics

### 7.2.1 Static semantic objects for Base

The context for the static semantics gives the bindings for any free identifiers, and is given by the grammar:

|                                                                                                 |                           |
|-------------------------------------------------------------------------------------------------|---------------------------|
| $C ::= V \Rightarrow T$                                                                         | initialized variable (C1) |
| $V \Rightarrow ?T$                                                                              | typed variable (C2)       |
| $S \Rightarrow \text{type}$                                                                     | type (C3)                 |
| $S \equiv T$                                                                                    | type equivalence (C4)     |
| $T \sqsubseteq T'$                                                                              | subtype (C5)              |
| $C \Rightarrow (RT) \rightarrow S$                                                              | constructor (C6)          |
| $\Pi \Rightarrow [(\text{gate}(RT))^*] (RT) [(\text{exn}(RT))^*] \rightarrow \text{exit}(T)$    | process identifier (C7)   |
| $\Pi \Rightarrow [(\text{gate}(RT))^*] (RT) [(\text{exn}(RT))^*] \rightarrow \text{guarded}(T)$ | process identifier (C8)   |
| $G \Rightarrow \text{gate}(RT)$                                                                 | gate (C9)                 |
| $X \Rightarrow \text{exn}(RT)$                                                                  | exception (C10)           |
|                                                                                                 | trivial (C11)             |
| $C, C$                                                                                          | disjoint union (C12)      |

where each identifier only has one binding.

The operations which we use on contexts are the “,” (disjunct union), “ $\odot$ ” (matching union), “;” (context over-riding), and “ $-$ ” (subtraction).

We shall write  $C_1 \odot C_2$  to denote the matching union on context. It is well defined only if the common names of  $C_1$  and  $C_2$  have the same bindings in both contexts. Its formal definition is:

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

We shall write  $C_1; C_2$  for context over-riding (with all the bindings of  $C_2$ , and any bindings from  $C_1$  not overridden by  $C_2$ ).

We shall write  $C - \{V_1, \dots, V_n\}$  for subtracting variables from the context (if  $V_1 \Rightarrow T_1, \dots, V_n \Rightarrow T_n$  and  $V_1 \Rightarrow ?T_1, \dots, V_n \Rightarrow ?T_n$  belongs to  $C$ , then they are removed).

In order to avoid many parentheses, “ $\odot$ ” has higher precedence than “;”, and “,”.

The semantics for modules uses *renaming* on contexts. Since renaming is a particular case of substitution, we write  $C[\sigma]$  ( $\mathcal{B}[\sigma]$ ) for the contexts that are obtained by applying the substitution  $\sigma$  in  $C$  ( $\mathcal{B}$ ).

Note that the grammar for record types overlaps with that of contexts. Whenever  $RT$  does not contain any occurrences of **etc**, we shall allow  $RT$  to range over contexts (for example in the type rule for sequential composition in Subclause 9.2.6).

### 7.2.2 Judgements on static semantics for Base

In this subclause, we describe judgements that define the static semantics for the base language.

**Behaviours** The static semantics is given by a series of judgements, such as

$$C \vdash B \Rightarrow \mathbf{exit}(RT)^1$$

meaning “in context  $C$ , behaviour  $B$  has result type  $(RT)$ ” or

$$C \vdash B \Rightarrow \mathbf{guarded}(RT)$$

meaning “in context  $C$ , behaviour  $B$  has result type  $(RT)$  and cannot exit initially”.

Besides, the following rules apply:

$$C \vdash B \Rightarrow \mathbf{exit}(RT)$$

$$C \vdash RT \sqsubseteq RT'$$

$$\hline C \vdash B \Rightarrow \mathbf{exit}(RT')$$

$$C \vdash B \Rightarrow \mathbf{guarded}(RT)$$

$$C \vdash RT \sqsubseteq RT'$$

$$\hline C \vdash B \Rightarrow \mathbf{guarded}(RT')$$

$$C \vdash B \Rightarrow \mathbf{guarded}(RT)$$

$$\hline C \vdash B \Rightarrow \mathbf{exit}(RT)$$

**Type expressions** Subtyping is a preorder:

$$\overline{C \vdash T \sqsubseteq T}$$

$$\frac{C \vdash T \sqsubseteq T' \quad C \vdash T' \sqsubseteq T''}{C \vdash T \sqsubseteq T''}$$

We write  $T \equiv T'$  for  $T \sqsubseteq T'$  and  $T' \sqsubseteq T$ . We will write:

$$C \vdash T_1 \sqcup T_2 \Rightarrow T \quad C \vdash T_1 \sqcap T_2 \Rightarrow T$$

whenever (up to  $\equiv$ )  $T_1$  and  $T_2$  have a least upper bound (respectively greatest lower bound)  $T$ .

**Record type expressions** The following judgements are used in the definition of the static semantics for record type expressions:

$$C \vdash RT \Rightarrow \mathbf{record}$$

$$C \vdash RT \sqsubseteq RT'$$

Subtyping is a preorder:

$$\overline{C \vdash RT \sqsubseteq RT}$$

$$\frac{C \vdash RT \sqsubseteq RT' \quad C \vdash RT' \sqsubseteq RT''}{C \vdash RT \sqsubseteq RT''}$$

We write  $RT \equiv RT'$  for  $RT \sqsubseteq RT'$  and  $RT' \sqsubseteq RT$ .

<sup>1</sup>**exit** is overloaded. The context in which it appears will indicate if we refer to it as a behaviour or as the result of a judgements.

**Value expressions** The following judgements are used in the definition of the static semantics for value expressions:

$$C \vdash N \Rightarrow T$$

$$\frac{C \vdash N \Rightarrow T \quad C \vdash T \sqsubseteq T'}{C \vdash N \Rightarrow T'}$$

**Record value expressions** The following judgements are used in the definition of the static semantics for record value expressions:

$$C \vdash RN \Rightarrow RT$$

$$\frac{C \vdash RN \Rightarrow RT \quad C \vdash RT \sqsubseteq RT'}{C \vdash RN \Rightarrow RT'}$$

**Patterns** The following judgement is used in the definition of the static semantics for patterns:

$$C \vdash (P \Rightarrow T) \Rightarrow (RT)$$

This judgement means that in context  $C$  it is possible to match the type of  $P$  with  $T$ , by means of the bindings expressed in  $(RT)$ .

**Behaviour pattern-matching**

$$C \vdash (BM \Rightarrow T) \Rightarrow \mathbf{exit}(RT)$$

$$C \vdash (BM \Rightarrow T) \Rightarrow \mathbf{guarded}(RT)$$

**General axioms**

$$C, J \vdash J$$

This axiom means that any binding  $J$  from a context  $C, J$  may be inferred.

### 7.2.3 Extended identifiers

The *extended identifier*  $e-id$  of an occurrence of an *identifier*  $id$ , is an extension of an identifier with the *scope* information of that identifier. The extended identifier  $e-id$  of an identifier  $id$  belonging to the following classes: Typ, Con, Fun, and Procl is a pair  $\langle scp, id \rangle$ . The scope  $scp$  of the identifier  $id$  is the name of the module, generic module, interface, or specification where it was defined or declared.

In the definition of semantic objects for the base language, all identifiers from classes Typ, Con and Procl are extended identifiers.